

© 2015 Qingkun Li

AN ENERGY-EFFICIENT HARDWARE SYSTEM FOR ROBUST AND  
RELIABLE HEART RATE MONITORING

BY

QINGKUN LI

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Electrical and Computer Engineering  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisers:

Professor Ravishankar K. Iyer  
Professor Zbigniew T. Kalbarczyk

# Abstract

Cardiac arrhythmia, one of the most common causes of death in the world today, is not always effectively detected by regular examinations, as it usually occurs infrequently and suddenly. Therefore, real-time, continuous monitoring of the heart rate is needed to detect arrhythmia problems sooner and prevent their severe consequences. To make continuous monitoring possible and give it widespread acceptance, a portable heart rate monitoring system must have three key characteristics: (1) accuracy, (2) portability, and (3) long battery life. Previous studies have focused on addressing these problems separately, either improving the accuracy of the monitoring algorithm or the efficiency of the underlying hardware.

This thesis proposes a robust and reliable heart rate monitoring system (RRHMS), in which both algorithm accuracy and hardware efficiency are considered. As a result, algorithmic optimizations are exploited to enable further hardware efficiency. In the RRHMS, robust heart rate monitoring is achieved by extracting heart rates from both electrocardiogram (ECG) and arterial blood pressure (ABP) signals and fusing them based on the signal qualities. Therefore, accurate heart rate data can be provided continuously, even when one signal is severely corrupted. Algorithmic optimizations are applied to merge the separate ECG and ABP processing steps into shared ones, which allows shared hardware modules and hence low-area (portable) hardware design. Also, an embedded hardware architecture framework is proposed for the design of the RRHMS hardware system. Coarse-grained functional units (FUs) can be easily added or removed in this framework, allowing for application-specific hardware optimization. Further, the application invariant properties are used to achieve low-overhead fault tolerance in the FUs to enhance reliability. Both ASIC and FPGA implementations of the RRHMS are able to accurately detect heart rates in real time while consuming only 1/2870 and 1/923 of the energy required by the Android implementation.

*To my family and friends, for their love and support*

# Acknowledgments

I would like to express the deepest appreciation to my advisers, Professor Zbigniew Kalbarczyk and Professor Ravishankar Iyer. Not only did they patiently impart to me the knowledge needed in my thesis work, but more importantly, they taught me how to conduct research, which I believe will have a profound influence in my future career.

I would like to thank Homa Alemzadeh, a senior PhD student in our research group, who started this project and collaborated with me on it. She has provided me with tremendous help to start my thesis work and is always willing to spend time discussing the direction and next next step as the project moves along. I would also like to thank Yangyang Yu, an undergraduate research intern in our group, for her great help in the implementation of RRHMS as well as the ASIC experiments. My sincere thanks also go to my other groupmates — Catello Di Martino, Arjun Prasanna Athreya, Cuong Manh Pham, Phuong Minh Cao, Zachary Estrada, Zachary Stephens, Subho Sankar Banerjee, Hui Lin, Safa Messaoud, Yogatheesan Varatharajah, Daniel Chen, Zhihao Hong, Dao Lu, and Key Whan Chung — for the enjoyable and insightful group retreats with them, as well as the birthday cake they prepared for me.

In addition, I would like to thank my roommate, Xiaobin Gao, who has been my classmate and friend for six years, for his company, help, and encouragement in both classes and life. He has sacrificed a large amount of time to take care of me when I was ill and had operations at the hospital.

Last but not least, I would like to express my deep gratitude to my parents, Junfeng Li and Caifeng Hu, and my girlfriend, Shan Liang, for their endless love, support, and encouragement. Although my parents are far away in China, a phone call to them at any time always brings me confidence and energy to face any difficulty. My girlfriend often provides good advice in my research, and always brings me endless happiness and motivation.

# Table of Contents

Chapter 1	INTRODUCTION . . . . .	1
1.1	Contributions . . . . .	6
1.2	Thesis Organization . . . . .	7
Chapter 2	BACKGROUND AND RELATED WORK . . . . .	8
2.1	Biomedical Signals . . . . .	8
2.2	Biomedical Monitoring Algorithm . . . . .	11
2.2.1	Preprocessing . . . . .	11
2.2.2	Peak/Onset Detection . . . . .	13
2.2.3	Feature Extraction . . . . .	15
2.2.4	Signal Quality Assessment . . . . .	17
2.2.5	Multi-Signal Analysis and Fusion . . . . .	19
2.3	Biomedical Monitoring Hardware . . . . .	23
2.4	Hardware Fault Tolerance . . . . .	29
Chapter 3	RRHMS HEART RATE ALGORITHM . . . . .	32
3.1	Algorithm Overview . . . . .	32
3.2	Peak Detection . . . . .	33
3.3	Signal Quality Evaluation . . . . .	37
3.4	Heart Rate Estimation and Fusion . . . . .	43
3.5	Shared Processing . . . . .	44
Chapter 4	RRHMS HARDWARE SYSTEM . . . . .	47
4.1	Hardware System Overview . . . . .	47
4.2	Functional Unit Design and Configuration . . . . .	49
4.3	MIPS Controller . . . . .	53
4.4	Robust Heart Rate Application Mapping . . . . .	56
Chapter 5	RRHMS FAULT TOLERANCE DESIGN . . . . .	59
5.1	Fault Model and Injection . . . . .	59
5.2	Hardware Coverage . . . . .	61
5.3	Fault Detection . . . . .	62
5.4	Fault Recovery . . . . .	66
5.5	Fault Tolerance Coverage Discussion . . . . .	68

Chapter 6	EXPERIMENTAL RESULTS . . . . .	72
6.1	Heart Rate Detection Accuracy . . . . .	72
6.2	Hardware Experiment Setup . . . . .	75
6.3	Baseline RRHMS Evaluation . . . . .	77
6.3.1	Comparison of Android, FPGA, and ASIC . . . . .	77
6.3.2	RRHMS Resource Utilization . . . . .	81
6.3.3	Runtime, Resource, and Power Breakdown . . . . .	81
6.4	Fault Tolerance Evaluation . . . . .	84
6.4.1	FDRU Overheads . . . . .	84
6.4.2	Fault Injection Methodology . . . . .	86
6.4.3	Fault Tolerance Coverage . . . . .	88
6.4.4	Discussion of Fault Coverage . . . . .	92
Chapter 7	CONCLUSION AND FUTURE WORK . . . . .	97
7.1	Conclusion . . . . .	97
7.2	Future Work . . . . .	98
7.2.1	Software/Hardware Partitioning . . . . .	98
7.2.2	Fault Behavior Analysis . . . . .	99
7.2.3	Functional Units Pipelining . . . . .	101
References	. . . . .	102

# Chapter 1

## INTRODUCTION

This thesis presents a Robust and Reliable Heart Rate Monitoring System (RRHMS) to provide portable and real-time monitoring at low energy consumption. The proposed RRHMS includes: (1) a robust heart rate detection algorithm with multiple signal analysis and fusion, and (2) a reliable embedded hardware design that efficiently runs the heart rate detection algorithm and is able to tolerate low-level hardware faults.

Continuous monitoring of heart rate is key to detecting cardiac arrhythmia problems, which affect more than 5 million Americans and result in more than 1.2 million hospitalizations and 400,000 deaths each year in the U.S. alone [1]. Arrhythmias may occur at any age, and some of them are barely perceptible until a problem occurs [2]. Moreover, arrhythmia problems are often characterized by suddenness and unpredictability. To realize the required continuous monitoring and receive wide acceptance, three qualities are key to heart rate monitoring systems: **accuracy**, **portability**, and **battery life**.

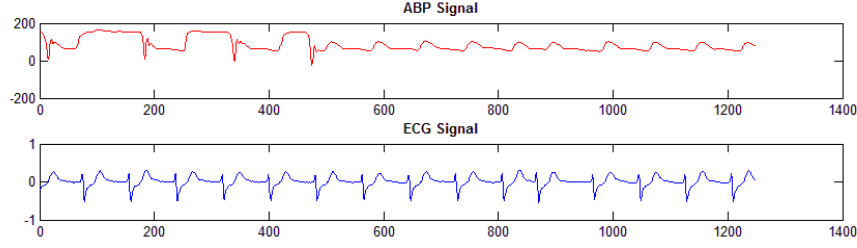
According to [3], the **accuracy** of the monitoring system depends on: (1) correctness and quality of the raw input signal collected from sensors, (2) adequacy of the signal analysis algorithm (e.g., removes signal noise, correctly extracts features, has no bugs, etc.), and (3) correctness of the underlying hardware that runs the monitoring algorithm. Addressing the first issue with better sensor technology is not the focus of this thesis. However, the proposed RRHMS considers the effect of sensor signal quality and is able to provide continuous monitoring in scenarios with low-quality signals.

*Impact of the Quality of Monitored Signals.* Traditionally, heart rate information is acquired by detecting peaks (location of the highest magnitude in the signal period) from the electrocardiogram (ECG) or arterial blood pressure (ABP) signals; arrhythmia is detected by thresholding on the heart rate. However, signal corruptions in ECG and ABP (such as noise, artifacts, missing data, etc.) often result in inaccurate detection of peaks and then to

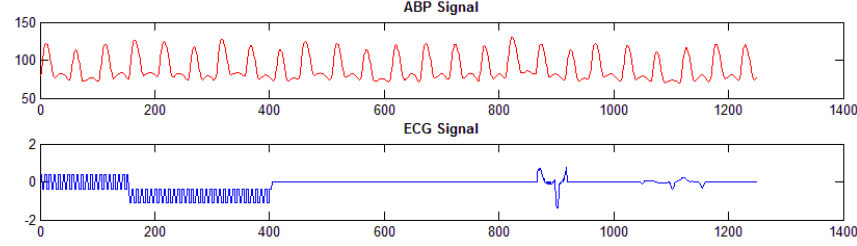


false diagnoses [4]. The large number of false alarms in intensive care units (ICUs) is an example of this problem. Studies [5, 6, 7] show that more than 80% of the alarms generated in ICU monitors are false and clinically insignificant due to corruptions in the sensor signals and the simple thresholding policy on a single signal. Although sophisticated signal filtering and analysis techniques [8, 9, 10] can alleviate this problem to an extent, they do not work when the signal corruption is severe. However, since noise and artifacts in different signals are weakly correlated [11], robust heart rate estimation is achieved in RRHMS by analyzing both ECG and ABP signals, as heart rate can be estimated from both of them. Figure 1.1 shows three segments of patient data from the MIMIC II database [12]. In Figure 1.1a, ABP signal is good while ECG signal is corrupted by sensor noise and disconnection, and during this period, the patient is experiencing the tachycardia problem. This problem cannot be detected if only ECG is being monitored. However, if the monitoring system considers both ABP and ECG signals, and if it detects that ECG is currently having low signal quality while ABP signal is good (Section 2.2.4 gives an overview of signal quality assessment methods), then it can detect the problem by ignoring ECG and using only ABP. Figure 1.1b shows the opposite case: the beginning part of ABP is noisy, but ECG is good. Similarly, during that period, the monitoring system needs to ignore ABP and use ECG to provide continuous heart rate monitoring. At last, both ABP and ECG are good in Figure 1.1c, where a strong correlation between the two signals for providing heart rate information is shown by the dotted lines. When both signals are good, the monitoring system can have high confidence in estimating the heart rate.

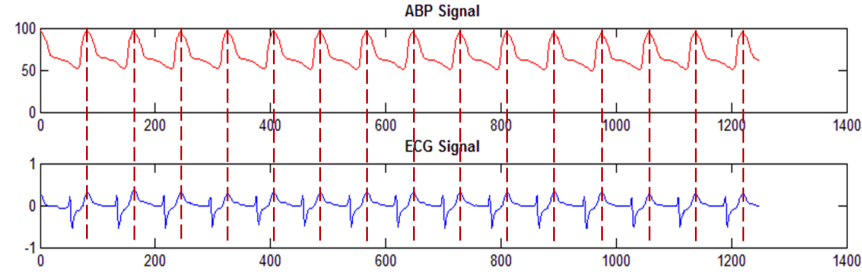
*Impact of Hardware Faults.* On the other hand, even if the monitoring algorithm is highly accurate, along with the scaling of transistor technology, hardware components become increasingly susceptible to low-level transistor faults that may propagate to cause the electronic systems to produce incorrect results, hangs, or crashes [13]. Transient faults or soft errors are the most common hardware faults, the rate of which is expected to have an 8% increase per logic state bit in each technology generation [14, 15]. For heart rate monitoring, this may cause the system to generate incorrect heart rate information or hang without notice. Therefore, enabling fault tolerance capability, especially tolerance of transient faults, is important (sometimes even life-critical) for heart rate monitoring systems. In addition, fault toler-



(a) ABP is good while ECG is corrupted (patient is experiencing tachycardia)



(b) ABP is noisy while ECG is good



(c) Both ABP and ECG are good (ABP and ECG peak locations are correlated)

Figure 1.1: Three scenarios of ABP and ECG signals from the MIMIC II database (patient a40050)

ance should not introduce large hardware area and power overheads (as with the traditional double and triple modular redundancy techniques), which jeopardize the portability and battery life of the monitoring system.

**Portability** and **battery life** are usually translated to hardware area and power consumption. As discussed above, accurate real-time monitoring of the heart rate requires concurrent recording and analysis of multiple biomedical signals (ABP and ECG) collected at relatively high sampling rates (up to 10 kHz [16]). Therefore, portable monitoring systems often face challenges in real-time processing of large amounts of biomedical signal data under tight area and power constraints. Commercial off-the-shelf embedded processors, such as the ARM microprocessor and DSP (digital signal processor), are not the best solutions to provide continuous and portable biomedical monitor-

ing. They are designed for general embedded applications with instruction-level optimizations, but they lack the application-specific optimizations for biomedical monitoring, and therefore, they do not offer the best performance and power efficiency for running biomedical monitoring applications. The comparison results in Section 6.3.1 between the proposed RRHMS and the Qualcomm Krait processor [17] (architecturally similar to ARM Cortex-A15) are proofs of this. In addition, higher power consumption drains the battery faster, and as a result, a bigger and heavier battery is required, which affects portability as well.

Existing studies and techniques have provided good solutions to the separate problems of monitoring accuracy and processing efficiency. For example, studies in [8, 9, 10] have focused on developing noise filtering and signal quality assessment techniques to obtain clean signals and the signal quality information needed for improving the analysis accuracy. Other works [18, 19] presented signal analysis algorithms for accurate peak detection of either ECG or ABP signals. The algorithms proposed in [11, 20, 21, 22] separately analyzed multiple signals, including ABP and ECG, and fuse the analysis results to detect robust heart rate and reduce false alarm rates in the detection of different cardiac arrhythmia problems. Moreover, Khatib et al. [16] explored using ARM and DSP embedded processors to optimize an autocorrelation-based ECG peak detection algorithm, and customized hardware modules were applied by [23, 24, 25] to achieve further performance and energy efficiency for previously developed ECG peak detection algorithms. *However, none of the previous works has considered addressing algorithm accuracy and processing efficiency issues at the same time by including multiple signal analysis and fusion techniques along with both software and hardware design and optimizations. As a result, some software/hardware optimization opportunities have been overlooked. In addition, none of the previous biomedical monitoring hardware research has considered the important feature of hardware fault tolerance.*

*Design Overview.* This thesis introduces RRHMS to address issues of both monitoring accuracy and processing efficiency in order to achieve accuracy, portability, and long battery life in heart rate monitoring. Figure 1.2 depicts the overview of the proposed heart rate monitoring system. In RRHMS, both ABP and ECG signals are analyzed and fused to achieve robustness. Unlike previous signal-fusion systems, such as [20, 21], which extract heart

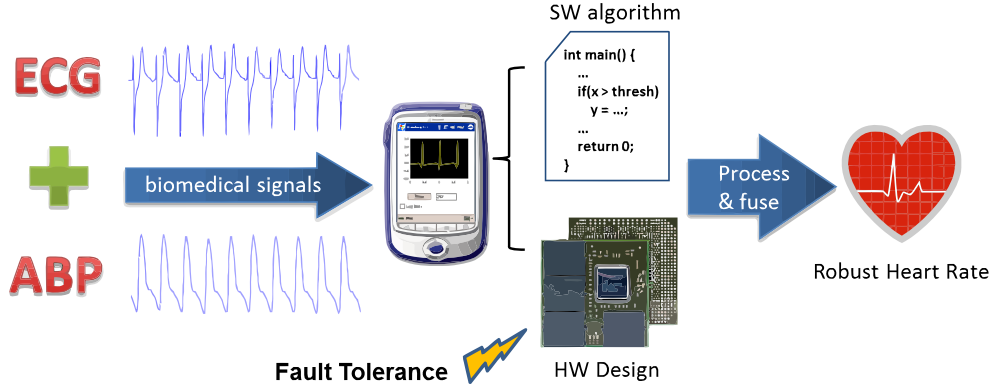


Figure 1.2: The proposed robust and reliable heart rate monitoring system

rate feature from ABP and ECG separately with different processing steps, RRHMS considers both algorithm and hardware system design. Algorithmic optimizations are applied to estimate heart rates in both signals with shared processing steps to allow hardware module sharing for efficient and low-area (portable) hardware design. After extracting heart rates from both signals, RRHMS evaluates their signal qualities and puts more weight (trust) on the heart rate information extracted from the signal with the higher signal quality. In addition, to efficiently support the processing steps of the heart rate detection algorithm, an embedded hardware architecture framework with coarse-grained configurable functional units (FUs) is proposed in which FUs can be added or removed in the architecture framework for application-specific hardware optimizations. The reliability of RRHMS is achieved by introducing a hardware fault tolerance technique that uses the application-specific invariant property [26] for low-overhead fault detection and recovery. The proposed fault tolerance technique is able to detect both permanent and transient hardware faults in the FUs and dynamically recover from transient faults in real time. *To the best of our knowledge, RRHMS is the first low-energy and fault tolerant hardware implementation for real-time robust heart rate estimation by multiple signal analysis and fusion.*

Experiments show that the algorithmically optimized peak detection algorithm is able to match the results of the originally separate ABP and ECG peak detection algorithms, and the applied fusion algorithm [21] provides continuous robust heart rate information even when one signal is corrupted. The proposed RRHMS is implemented in both the ASIC design using the Synopsys Design Compiler [27] and on the FPGA platform with the Virtex 5

ML507 board. The results of the implemented hardware on both platforms are compared with MATLAB simulations to ensure implementation correctness. The same algorithm is also implemented as an Android application on a Nexus 7 tablet (equipped with the Qualcomm Krait processor) for comparison. The simulation results show that, compared with Android, both ASIC and FPGA implementations achieve better runtime performance (20.6 and 13.7 times speedup, respectively) at much lower power consumptions (1/139 and 1/67 of Android’s power, respectively). Moreover, the proposed fault tolerance technique enhances system reliability by increasing the correct output percentage under injected hardware faults. It is able to reduce more than 55% of incorrect outputs and system failures under all tested fault rates. In addition, the proposed fault tolerance mechanism has only about a 15% area (resource) overhead in the hardware logics, and during normal monitoring when no fault occurs, it only introduces about a 34% power overhead (due to fault detection checking) and does not incur any performance overhead.

## 1.1 Contributions

The specific contributions of this thesis are summarized as follows:

- We apply algorithmic optimizations to ECG and ABP heart rate detection algorithms developed by [18] and [28], respectively. The optimizations enables shared processing steps between the two originally separate processing flows. As a result, some hardware modules that are used to efficiently support ECG and ABP processing are shared. The shared modules account for about 45% of the hardware logics, effectively cutting the hardware area in half.
- We propose an embedded hardware architecture framework with coarse-grained configurable functional units (FUs) to perform energy-efficient computations. The FUs can be configured within a few cycles to switch between ABP and ECG analysis or between different patients. Additionally, FUs are designed following a framework design template, which gives them the same interfaces. This makes it easy to add or remove FUs in the architecture framework to achieve application-specific hardware optimizations for the target embedded application.

- We propose and design a low-overhead hardware fault detection and recovery engine that monitors the activities of the FUs and applies invariant checking and heartbeats to detect hardware faults. Upon detection of faults, the corresponding FU is reset and re-executed by the fault detection and recovery engine in real time to dynamically recover from the transient fault. Permanent faults, however, can only be detected, not recovered from.
- We implement the RRHMS prototype both in the ASIC design and on the FPGA platform to evaluate runtime performance, power and energy consumption, and fault tolerance coverage and overheads of the proposed fault detection and recovery engine. ASIC is the target platform for the final product of RRHMS, while in the FPGA implementation, the proposed hardware architecture is evaluated as the framework to support the target embedded application.

Homa Alemzadeh and Yangyang Yu are my collaborators in this thesis work. In particular, Alemzadeh has helped in (1) the proposal and implementation of the multiple signal (ECG and ABP) analysis and fusion algorithm, (2) the algorithmic optimizations to merge the separate ECG and ABP analysis algorithms, and (3) the design of the RRHMS hardware architecture framework. Yangyang has helped with the implementation of the coarse-grained FUs and the ASIC synthesis of the RRHMS.

## 1.2 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 provides the background of biomedical signal monitoring as well as the work in this field that motivates the development of RRHMS. Chapter 3 introduces each processing step of the RRHMS heart rate detection algorithm, including the applied algorithmic optimization. Chapter 4 discusses in detail the proposed baseline RRHMS hardware system without fault tolerance features, followed by discussion of the proposed low-overhead fault tolerance design in Chapter 5. Chapter 6 introduces the experiments performed and the results obtained in evaluating RRHMS. The thesis concludes and future work is described in Chapter 7.

## Chapter 2

# BACKGROUND AND RELATED WORK

In this chapter, we provide an overview of the background of biomedical monitoring. First, common biomedical signals used in the monitoring are briefly introduced, followed by discussion of software and hardware approaches previously proposed to analyze and extract features from those biomedical signals. Previous work provides us with valuable insights of the characteristics of different biomedical signals, as well as useful analysis of the efficiency of different hardware systems for biomedical signal processing. However, the related previous work, such as the work in signal beat detection and signal fusion, focuses on developing and applying separate signal processing steps to extract features from different signals. Additionally, previous biomedical hardware work only uses the hardware capability for efficient computations of previously developed algorithms. None of the work considers software optimizations for efficient hardware design or the fault tolerance capability of the hardware.

### 2.1 Biomedical Signals

Biomedical (or physiological) signals are the signals produced by the body during the functioning of various physiological systems [29]. Information about the state of the physiological system can be extracted from the corresponding biomedical signals. The common biomedical signals being monitored and checked in hospitals are: electrocardiogram (ECG), arterial blood pressure (ABP), peripheral capillary oxygen saturation (SpO<sub>2</sub>), and respiratory rate. These four commonly used signals are the ones typically studied in the literatures of biomedical monitoring algorithms. Figure 2.1 shows an example segment of each of these four biomedical signals. Each signal is collected by a different bio-sensor. Table 2.1 lists the sensors used to collect

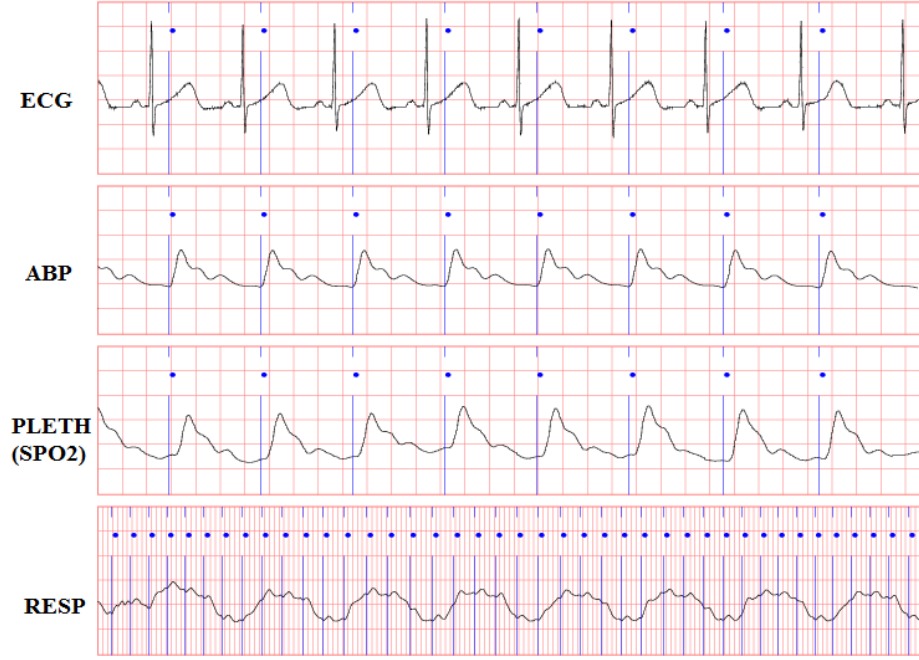


Figure 2.1: Commonly used biomedical signals (from the MIMIC database)

Table 2.1: Bio-sensors used to collect different biomedical signals

Biomedical Signal	Sensor	Working Principle
ECG	ECG electrodes	Detect the rise and fall of voltage caused by heart muscle depolarizing between the two electrodes placed at either side of the heart.
ABP	Pulse oximeters with pressure sensor	Control and measure counter pressure to keep constant finger blood volume.
SpO2	Pulse oximeters	Measure and calculate the emitted light absorbed by the finger tip.
Respiratory rate	1. Capnography or 2. Pneumatograph	1. Monitor the concentration or partial pressure of carbon dioxide ( $\text{CO}_2$ ) in the respiratory gases. 2. Record velocity and force of the chest movements during respiration.

each signal, along with a brief description of each sensor’s working principles.

**ECG signal** records the electrical activity of the heart. Up to 12 leads of ECG signals can be obtained by placing electrodes at different places on the body [30], and they together provide accurate ECG analysis [16].



ECG signals are often used to detect various cardiac diseases and problems [31, 32, 33], including coronary heart disease, pericardium disease, cardiac arrhythmias, etc. **ABP signal** indirectly indicates information about the health state of the heart and blood vessels. Low blood pressure, or clinical shock, usually indicates a medical emergency situation in the intensive care unit or during the surgery operation, and high blood pressure is a sign of chronic condition hypertension [34]. Heart rate and cardiac output (which is a key parameter in assessing circulatory function) can be estimated from ABP waveforms as well [35]. **SpO2 signal** measures the blood oxygen saturation level, which is the percentage of hemoglobin in the blood that is saturated with oxygen, and indicates if sufficient oxygen is being supplied to the body, especially to the lungs [36]. **Respiratory rate** is the number of breaths taken within a set amount of time. Monitoring of respiratory rate is important in postoperative care, because some operations, like abdominal operations, might cause central or obstructive apnoea [37, 38]. In addition, Fieselmann et al. [39] show that monitoring respiratory rates is helpful in reducing the chance of cardiopulmonary arrest.

Processing and operations on those biomedical signals follow the same general operation flow depicted in Figure 2.2. First, the raw biomedical signals are collected from the body using bio-sensors (Table 2.1). After collection of the raw signals, signal processing methods are applied to extract useful features from the signals to identify patient problems and make diagnostic decisions (for example, the QRS complex feature extracted from ECG is used in [22, 40, 41] to identify asystole, ventricular tachycardia, coronary artery disease, and so on). In addition, the processing of the raw biomedical signals

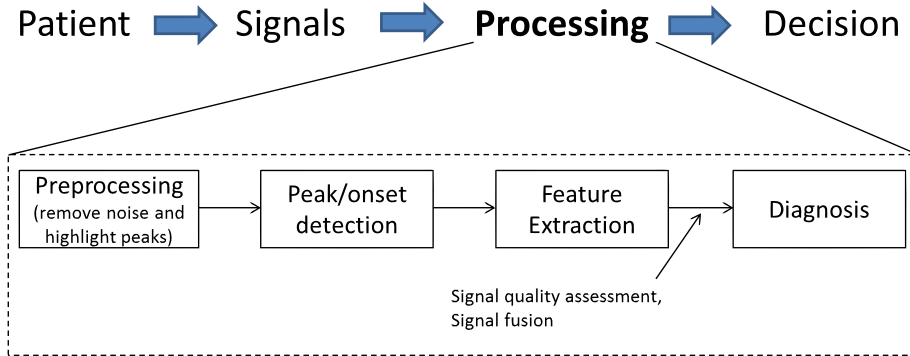


Figure 2.2: General operation flow of biomedical signals

also follows the four general processing steps as shown in Figure 2.2 (in the dash-line box): preprocessing, peak/onset detection, feature extraction, and diagnosis. (Section 2.2 discusses each step in detail.)

This thesis focuses on heart rate monitoring. ECG, ABP, and SpO2 signals all contain information about heart rate, as the electrical activity recorded by ECG, the blood pressure waveform in ABP, and the pulse oximeter’s plethysmograph for SpO2 calculation are periodic with the heart beats. Only ECG and ABP signals are used in this thesis because we found few patient SpO2 signals during the period in which both ECG and ABP signals were valid in the MIMIC II database. However, since SpO2 has waveform shapes similar to ABP, we believe the RRHMS heart rate detection and fusion algorithm, as well as the RRHMS hardware system, can be used to efficiently process the SpO2 signal and fuse the heart rate extracted from it as well. If SpO2 is included in RRHMS, the heart rate estimation would be more robust, for it could then tolerate severe signal corruptions in up to two out of the three signals.

## 2.2 Biomedical Monitoring Algorithm

Numerous algorithms to extract and analyze features from biomedical signals have been proposed and studied, especially with ECG and ABP signals. Different works focus on the different processing steps shown in the dash-line box of Figure 2.2. This section discusses in detail the purpose of each processing step and the techniques used or developed for each step in the literature, along with a brief introduction to the technique applied for each processing step in RRHMS. We were motivated by the literature to develop them, and at the same time, we considered the effect of the processing algorithm development on the hardware design and optimization.

### 2.2.1 Preprocessing

The raw biomedical signals obtained from the bio-sensors often contain a lot of noise, making the features difficult to extract correctly from the signal. So the preprocessing stage is applied to (1) reduce the noise in the signal (to maximize the signal-to-noise ratio), and (2) highlight the interested feature

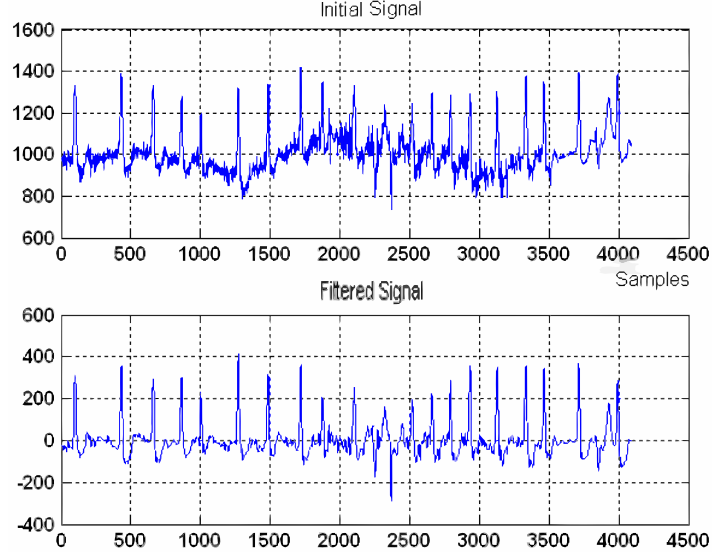


Figure 2.3: Noise in ECG signal and filtered ECG signal [42]

information.

Figure 2.3 illustrates an example of ECG signal before and after noise removal. Signal noise is normally caused by different types of artifacts and interferences in the sensors [43, 44], such as power line interference, electrode contact noise, motion artifacts, etc. Digital filters, such as low-pass, high-pass, and median filters, are the most commonly used method for signal noise removal, as used in [45, 46, 47, 48]. To simplify the digital filter design and realize fast computation on small computers, Lynn [8] proposed the “fast designs” for a class of digital filters for biomedical signals, including the recursive and non-recursive low-pass, high-pass, band-pass, and band-stop filters. In addition, digital differentiation and squaring are used in [18, 49] to further remove noise and highlight peak locations for ECG QRS detection. More recent studies [10, 50, 51, 52] use adaptive filters (with electrode-skin impedance as the reference signal) and wavelet-based techniques for the noise removal in ECG signals. The extended Kalman filter is applied on the ABP signal in [53] for signal artifact smoothing. RRHMS applies the digital filter method to remove noise in ABP and ECG signals. It is designed following the method developed by [8], based on its efficiency and simplicity of hardware implementation.

As for feature highlighting, slope sum has been used in [20] to highlight and smooth the rising portion of the ABP signal. It has an effect similar to the moving-window integration used in [18]. But since the two works

[18, 20] focus on the processing of ABP and ECG signals, respectively, two separate feature highlighting methods with similar effect were developed. Since RRHMS considers the processing of both signals, as well as hardware optimization, we optimized by algorithmically merging these two separate processing steps into a single step that highlights the rising portions of both ABP and ECG signals. As a result, RRHMS requires only a single, shared hardware module for efficient feature highlighting.

### 2.2.2 Peak/Onset Detection

Many biomedical signals, such as ECG, ABP, SpO<sub>2</sub>, etc., are periodic with the heart beat. Features extracted within a beat (e.g., systolic/diastolic blood pressure values, ECG QRS shape, beat-to-beat interval, etc.) and their changes from beat to beat are the foundations of detecting diseases and problems. Therefore, it is critical to accurately identify the beat or period of the biomedical signals to accurately extract features used for diagnosis [54, 55]. Signal peak/onset detection techniques were developed for this purpose, and three detection techniques are commonly used: (1) threshold-based detection, (2) frequency analysis detection, and (3) template matching detection.

Many works [18, 28, 56, 57, 58] apply the threshold-based peak/onset detection method by thresholding on the signals in the time domain, combined with local maximum and minimum searching. For example, Zong et al. [28] designed the ABP onset decision rules in this way. First, adaptive thresholding is applied on the slope summed signal (used to highlight the rising portion of the raw ABP signal) to detect the onset. Then, local maximum and minimum searching is applied around the onset point to decide whether to accept the detected onset based on the difference between the maximum and minimum values. In [18], the ECG peak or QRS detection algorithm uses two thresholds to detect the ECG QRS complex. The higher threshold is used for the first-time analysis of the signal, while the lower threshold is applied if no QRS is detected within a certain amount of time; this enables back-searching for the QRS complex. The threshold values are usually dynamically updated to adapt to the on-the-fly signal changes. This is done according to different rules and conditions, such as signal quality and noise

level. A common method used in the literature to update the threshold values is to use the weighted sum function. When a peak is detected, the threshold values are updated by applying the weighted sum to the old threshold values and the new values that come from the newly detected peak.

Frequency domain analysis, such as filter bank, power spectrum, and energy analysis, is another popular method to detect beats. Power spectrum analysis has been applied as an intermediate step in [59] for beat detection of the pressure signals, such as the intracranial pressure, ABP, and SpO2 signals. Pachauri and Bhuyan [19] applied the window-based energy analysis technique, combined with the amplitude and interval thresholds, to detect peaks in the ABP signal. In addition, the filter bank based ECG beat detection algorithms have been introduced in [54, 60, 61]. Generally, the filter bank based algorithm involves (1) decomposing the signal into frequency subbands, (2) processing the subbands according to the application, and (3) reconstructing the processed subbands for further analysis. The frequency domain analysis methods use the fact that signal peaks are usually concentrated in a different frequency range from the signal noise and other uninteresting segments. For example, the energy of ECG peaks usually concentrates around the frequency of 40 Hz [61]. As a result, the ECG peaks are detected in the filter bank techniques by analyzing the corresponding frequency subband.

In addition, the template matching method is used in [16, 62] for ECG beat detection. Template matching evaluates the similarity between the input signal in with a template signal period by performing auto-correlation or cross-correlation calculations. The template signal period is predefined or trained, and the calculated local maximum correlation value indicates the detection of a signal period (beat) in the input signal.

Even though the previously proposed beat detection algorithms obtain good results with high accuracy, they are developed for specific signals (e.g., some algorithms are only developed for ECG, while others are developed for ABP). Therefore, if both ABP and ECG beats need to be detected, separate beat detection algorithms have to be used for each type of signal. This is not a big problem if a general purpose processor is applied as the platform for the processing involved in beat detection. However, if application-specific hardware is used for highly efficient and portable processing, separate algorithms require separate coarse-grained hardware modules, and therefore

would occupy more hardware area and consume more power. To solve this problem, as in the case if the preprocessing steps, we developed a single peak detection algorithm in RRHMS by applying algorithmic optimizations to merge (1) the threshold-based ABP onset detection algorithm proposed in [28], and (2) the ECG QRS complex detection algorithm (threshold-based) developed by [18]. We chose the threshold-based detection algorithm, because it can achieve good runtime and accuracy performance with less complicated hardware design. On the other hand, the frequency-based analysis is computation-intensive and requires complicated hardware modules to perform transformations between time and frequency domains. Further, the template matching method does not adapt well to detecting nonstationary patterns (its accuracy decreases when the signal pattern changes).

### 2.2.3 Feature Extraction

After the beat is detected, features can be extracted from the signal period. At this stage, many diseases and problems can be detected by simply thresholding on their corresponding feature values. For example, hypotension and hypertension problems are detected by thresholding on the systolic and diastolic blood pressures, where systolic blood pressure is the peak ABP value of the period and diastolic blood pressure is the valley ABP value of the period. The heart rate feature is computed with the beat-to-beat interval, which is the interval between detected beats, and cardiac arrhythmias are detected by thresholding on the heart rate. In addition, the ABP pulse pressure (the difference between systolic and diastolic pressures), ECG beat shape, ECG ST and QT intervals, and other values contain information about a person’s cardiac health [63, 64, 65].

The above features are directly extracted from each beat, so we call them first-level features. However, simply thresholding on the first-level features often results in a large number of false alarms. This is because first-level features tend to be sensitive to signal noise or artifacts. Even with sophisticated preprocessing techniques, some noise may not be completely eliminated, which may result in incorrect first-level feature extractions and therefore false alarms. On the other hand, although the alarms generated by some sudden changes in the signals (that quickly come back to normal) are techni-

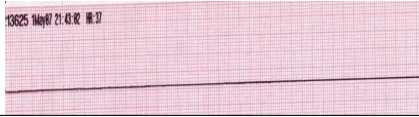


cally true alarms in the beat period where the change happens, those alarms are often clinically insignificant. The observation has been made in [66] that “rapid changes in the readings that swiftly return to normal are more likely non-physiological than those which are persistent and form a trend.” In addition, the similar claim has been made in [67] that “false alarms tend to occur fleetingly, while true alarms tend to develop more slowly.” Therefore, many methods do not rely only on first-level features to make diagnostic decisions but use second-level features as well.

Second-level features (defined by us) are features that are computed based on first-level features but are usually more robust to random noise and less sensitive to sudden non-physiological signal changes. For example, a second-level feature can be obtained by the statistical computation of first-level features within a time interval, such as the average, maximum, or minimum. In [66], the difference between the average heart rate in the current minute and that of three minutes ago is used to detect any change or trend in the heart rate, which helps filter false alarms. In [67], first-level features, such as ECG heart rate, respiratory rate, ABP mean and systolic pressures, are analyzed by windows of 10, 20, and 45 seconds. Within each interval, second-level features such as the mean, median, maximum, minimum, or standard deviation are computed for each first-level feature. The second-level features then are used by a machine learning algorithm to discover medical events, such as decreases in blood pressure, that require clinical attention. Seventeen-second ABP and ECG waveform segments are analyzed in [22], within which maximum, minimum, and mean heart rates are computed to reduce false alarms in detecting critical arrhythmias. Apiletti et al. [68] use the moving average on the first-level features of systolic and diastolic pressures, heart rate, and SpO2 to obtain second-level features for assessment of health status.

In addition, some problems can be detected only by looking at second-level features. For example, some current and potential cardiac diseases are undetectable using only the first-level heart rate feature but can be detected through analysis of the second-level feature of heart rate variability (HRV) over time [69].

To achieve robust heart rate monitoring, RRHMS calculates a second-level heart rate feature using the average heart rates of 10-second windows. This is key to accurately detecting several kinds of cardiac arrhythmia problems as well as to correctly analyzing the HRV. Table 2.2 lists some critical cardiac

Table 2.2: Arrhythmia problems that can be detected with RRHMS

Arrhythmia Type	Definition	Example Pattern
<b>Asystole</b>	No cardiac output for 4 seconds	
<b>Extreme Bradycardia</b>	Resting heart rate below 40 bpm	
<b>Extreme Tachycardia</b>	Resting heart rate above 140 bpm	

arrhythmias that can be detected with heart rate monitoring, along with a brief problem description and example signal pattern of the corresponding problem. The window interval chosen for use in RRHMS is large to minimize the effect of signal noise and non-physiological changes. Also, 10 seconds is the maximum delay allowed to meet the real-time arrhythmia detection criterion of AAMI-EC-13 Cardiotach Standard [22].

#### 2.2.4 Signal Quality Assessment

Some signal noises, artifacts, and abnormalities (e.g., those due to sensor movement or disconnection) can hardly be removed or fixed by signal filtering techniques. Figure 2.4, reproduced from [9], shows an example of a clinical ABP signal with abnormal regions, where the abnormalities are caused by sensor movements, not by problems in patient. When such an abnormality occurs, false alarms are very likely. Therefore, signal quality assessment techniques are developed to help identify the regions of the biomedical signals where noise or abnormality occurs and thus to further improve the accuracy of the signal analysis and diagnosis.

By observing and studying the nature of the normal signals, as well as the characteristics of different kinds of noise and artifacts, previous work [9, 20, 21, 70, 71] has developed various methods to evaluate signal quality and identify signal noise level. Sun et al. [9] flag the abnormal beats in the ABP signal by thresholding on the first-level features extracted from the beat, including systolic, diastolic, mean, and pulse pressures, beat duration,



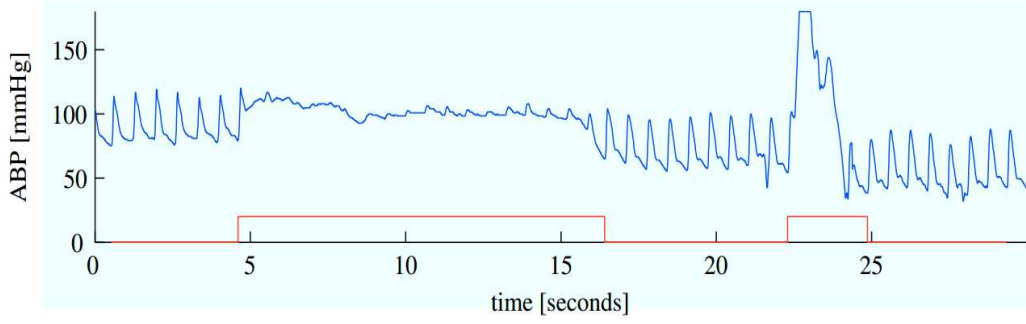


Figure 2.4: ABP waveform with abnormal regions [9]

and their differences between consecutive beats. Thresholds are set based on normal physiological ranges. For example, systolic blood pressure never exceeds 300 mmHg, so if an ABP beat has a peak value larger than 300, the beat is flagged as abnormal. Fuzzy logic is used in [20] to assess the quality of ABP beats by measuring how much the features of a detected beat deviate from the model features. Model features are trained using normal ABP signals and averaging the feature values extracted from each beat in the training period. Zong et al. [20] compute additional APB beat features based on those used in [9], including maximum positive and negative pressure slope, maximum up-slope duration, and so on. Kurtosis and spectral distribution analysis are applied by [21] to evaluate the signal quality of the ECG signal. Kurtosis is the fourth standardized moment and measures the peakedness of a distribution. A low kurtosis value in an ECG signal indicates the low-frequency baseline wander noise and the high-frequency power-line interference noise. Spectral distribution analysis relies on the fact that ECG peaks and noise are concentrated in different frequency bands. So the energy ratio of the frequency band containing ECG peaks indicates the quality of the ECG signal. Besides kurtosis, He et al. [70] use the variance index as an extra measure of ECG noise and abnormality level. In [71], the area differences between successive ECG beats, including adjacent beats, every other beat, every third beat, etc., are calculated to generate the statistical distributions used to assess ECG signal quality.

RRHMS evaluates the signal quality of both ABP and ECG and uses this information in fusing the heart rates extracted from the two signals. This minimizes the chance of incorrect heart rate estimations caused by signal noise or abnormality that cannot be fixed in the preprocessing step. RRHMS

signal qualities are calculated based on the previously developed methods [9, 20, 21].

### 2.2.5 Multi-Signal Analysis and Fusion

The fact that the causes of noise and artifacts in different bio-sensors are different and independent [11], inspired proposal of multi-parameter biomedical signal analysis and fusion methods to:

- (1) suppress false alarms generated by a single signal by exploiting the relationship between different sensor signals,
- (2) improve the accuracy of disease and problem detection by fusing the features extracted from different signals, and
- (3) provide robust and continuous monitoring, even when some signals are erroneous or corrupted, by using the redundant information in different signals.

The techniques used to fuse the analysis of different signals can be categorized as: (1) ***rule-based fusion***, which establishes rules and thresholds to fuse the results of different signals, (2) ***numeric-based fusion***, which fuses the features extracted from different signals with numeric calculations, such as weighted sum (or weighted average), and (3) ***machine learning-based fusion***, which combines the features acquired from different signals into the feature vectors and applies machine learning techniques to train and classify for specific problem detection. Table 2.3 lists some fusion techniques in each category.

The rule-based fusion method is applied by [22] and [72], using ABP and PLETH signals respectively, to reduce false alarms of arrhythmia in the ECG monitors used in intensive care units. The approach is to analyze the other signal (ABP and PLETH) whenever the ECG monitor raises an arrhythmia alarm, since it should contain redundant information. If the arrhythmia problem indicated by the ECG monitor's alarm is also detected in the redundant signal, the ECG alarm is accepted, otherwise it is rejected. Using the same rule-based method but in an opposite manner, Zong et al. [20] try to suppress the false alarms generated by the commercial ABP monitors by incorporating the ECG signal in computing the signal quality of ABP beats based on some rules and thresholds, such as thresholding on the ECG-ABP

Table 2.3: Previous multi-signal analysis and fusion work

Signal Fusion Works	Application Purpose	Raw Signal	Features	Technique Description
<b>Rule-Based Fusion</b>				
Aboukhalil et al. [22]	Reduce false alarms in ECG monitors by using ABP signal. Alarms include: - Asystole - Extreme bradycardia - Extreme tachycardia - Ventricular tachycardia	- ABP - ECG	- Heart rate - Beat-to-beat interval - Max. interval within a window - Min. interval within a window - Average heart rate in a window - Signal abnormality index [20]	- Reject ECG alarms by thresholding on the features extracted from ABP signal within the analysis window. - Design different threshold rules for different life-threatening alarms listed in the application purpose cell.
Zong et al. [20]	Reduce false alarms in the commercial ABP monitor by using ECG signal.	- ABP - ECG	- Heart rate - Systolic, diastolic, mean, and pulse blood pressures - Max. positive pressure slope - Max. negative pressure slope - Max. up-slope duration - Max. duration above threshold - ECG-ABP peak delay time	- Calculate both signal qualities of ABP and ECG. - Apply ECG signal to adjust the beat qualities computed for ABP signal. - Use the adjusted ABP signal quality as criteria to suppress alarms generated by ABP monitor.
Deshmane [72]	Suppress false alarms generated by the ECG monitor.	- ECG - PLETH	- Heart rate - Beat-to-beat interval - Max. interval within a window - Min. interval within a window - Average heart rate in a window - Signal quality of PLETH	- Suppress ECG alarms by thresholding on the features extracted from PLETH within the analysis window. - Design the threshold rules and alarm reduction framework similar to [22].
<b>Numeric-Based Fusion</b>				
Li et al. [21]	Provide robust heart rate estimation.	- ABP - ECG	- Heart rate - ABP and ECG signal qualities obtained by applying previously developed methods [9, 20, 70, 71]	- Apply Kalman filter on the ABP and ECG heart rates. - Obtain robust heart rate by applying weighted sum on the two estimated heart rates based on the signal qualities and Kalman residues.
Ebrahim et al. [11]	Provide robust heart rate estimation.	- ABP - ECG - PLETH	- Heart rate	- Identify erroneous sensor estimates by analyzing: a) consensus between sensors, b) comparison between current measurement and the predicted value, and c) physiologic consistency of the estimates. - Fuse sensor heart rates by weighted sum, excluding the erroneous estimate.
Apiletti et al. [68]	Propose frame work to classify health severity levels.	- ABP - ECG - PLETH	- Heart rate - Systolic and diastolic blood pressures - SpO2	- Compute risk component for each signal feature, such as sharp changes, long-term trends, and distance from normal behaviors (similar to second-level feature computations). - Estimate health severity level by applying weighted sum on all computed risk components.
(Continued on next page)				

Table 2.3 (cont.): Previous multi-signal analysis and fusion work

Signal Fusion Work	Application Purpose	Raw Signal	Features	Technique Description
<b>Numeric-Based Fusion - continued</b>				
Kannathal et al. [73]	1. Detect: - left ventricular failure - right ventricular failure - pulmonary oedema 2. Derive patient deterioration index.	- ABP - ECG - PLETH - RESP	- Heart rate - systolic, diastolic, and mean blood pressures - SpO2 - respiratory rate	- Apply fuzzy logic function with the input of quantized feature values to calculate the fuzzy probability of each failure. - Use weighted sum for each failure probability to obtain the deterioration index.
<b>Machine Learning-Based Fusion</b>				
Biosign [74]	Identify adverse trends in the patient health status and provide early warning of changes.	- ABP - ECG - PLETH - RESP - Temperature	- Heart rate - Systolic blood pressure - SpO2 - Respiratory rate	- Apply k-means clustering on the feature vectors to select 500 cluster centers. - Use Parzen window with the 500 prototype patterns as kernels to estimate the unconditional probability density function, which is used to derive the patient status index.
Tsien [67]	Detect medical event, such as ECG lead apnea, false alarms due to patient motion, blood pressure decrease, etc.	- ABP - ECG - PLETH - RESP	- Heart rate - Systolic and mean blood pressures - SpO2 - Respiratory rate	- Apply supervised machine learning method (decision tree) to classify the feature vectors for the detection of different medical events with the correspondingly trained models, where feature vectors are composed of the computed second-level features (min., max., median, etc. of a window).
Li and Clifford [75]	Suppress false arrhythmia alarms to improve detection accuracy.	- ABP - ECG - PLETH	- Heart rate - Systolic, diastolic, mean, and pulse blood pressures - SpO2 - Area difference of beats	- Use a genetic algorithm to select pertinent features from a large set of feature pool (second-level features computed within window intervals, such as maximum, minimum, median, variance, gradient, etc.). - Apply relevance vector machine to train and classify the selected feature vectors to improve detection accuracy.

beat delay time. The alarm generated by the the ABP monitor is judged by the ECG-adjusted ABP signal quality for acceptance. So rule-based, multi-signal analysis is not strictly a fusion method, as it does not actually fuse any features. It just uses other signals to judge the alarm raised by a single signal.

Work in robust heart rate detection [11, 21] and patient health status identification [68, 73] apply the numerical-based fusion method. In [11, 21], the robust heart rate is obtained by computing the weighted sum of the heart rates estimated from each sensor signal. Ebrahim et al. [11] first identify artifacts or erroneous sensor estimates and excludes them from the process

of the weighted sum. The sensor to be excluded is identified using the three metrics listed in the corresponding technique description cell in Table 2.3. [21] uses the previously developed signal quality assessment methods and proposes a novel use of the Kalman filter to compute the weights of different sensor signals. Risk components, such as sharp changes, long-term trends, and distance from normal behaviors, for each signal feature are quantified in [68]. The overall health severity level is derived by applying the weighted sum of the risk levels calculated using the risk components. In addition, Kannathal et al. [73] apply fuzzy logic functions to obtain the fuzzy probabilities of different cardiac failures. It uses a weighted sum with all the failure probabilities to derive a patient deterioration index to detect clinical status changes. The numeric-based fusion method is simple and flexible, as the features obtained from different sensor signals are easily fused through simple numerical calculations (yielding the weighted sum) and the signal weights can be adjusted using knowledge of the signals, such as their qualities and noise levels.

Machine learning-based fusion has been used for detecting broad medical conditions. In this approach, all the features extracted from different signals are grouped in feature vectors. Machine learning techniques are then applied to cluster or classify the vectors to detect various problems. For example, [74] uses k-means clustering with the features of heart rate, systolic blood pressure, SpO2, respiratory rate, and temperature to select 500 cluster centers. It then applies the Parzen window to derive a patient status index that can provide early warning of the patient status changes. Tsien [67] extracts 120 features from four raw signals and applies the supervised learning method to classify the feature vectors for different medical events with correspondingly trained models. Similarly, 114 signal features are extracted in [75], and a genetic algorithm is applied to select useful features from the large feature pool. The selected features are input to a trained relevance vector machine that classifies the feature vectors for detection of different arrhythmia problems. The works that apply machine learning-based fusion methods can obtain good results but have very high computation complexity, especially when on-line training is needed. Therefore, they may not be applicable in real-time monitoring when online model training is needed to adapt to signal changes or for patient-specific monitoring.

Multi-signal analysis and fusion are applied in RRHMS to provide robust and continuous monitoring. The previous work provide us with insights into different ways to fuse the analysis of multiple signals. We applied the numeric-based fusion method used in [21] to fuse the heart rates acquired from the ABP and ECG signals. We chose this method for its simplicity and flexibility when implemented in portable hardware for patient-specific monitoring in real time. However, the previous multi-signal analysis and fusion efforts only focus on the fusion itself; they acquire the raw features (such as heart rate) directly from the monitoring machines or calculate them using the previously developed noise filtering and beat-detection methods (different methods are used to process different signals with separate processing flows). In contrast, RRHMS deals with the whole computation, from signal preprocessing, to beat detection, to feature extraction and fusion, as well as the hardware design for real-time portable monitoring. So RRHMS can optimize the processing of separate signals for portable hardware design by developing shared processing steps for multiple signals, which in turn enables shared hardware modules. This takes us beyond the scope of the previous work in signal fusion.

## 2.3 Biomedical Monitoring Hardware

To enable efficient biomedical signal processing and analysis in real time, biomedical monitoring hardware has been developed. Based on the applied hardware platform, the work dealing with hardware is classified into four categories: (1) embedded-processor (DSP and ARM) based, (2) coarse-grained reconfigurable array (CGRA) based, (3) FPGA-based, or (4) ASIC-based. Table 2.4 compares the hardware work in each category.

Embedded-processor based approaches [16, 76, 77] propose system-on-chip (SoC) solutions to optimize the biomedical signal processing algorithms on off-the-shelf embedded processors, such as DSP and ARM. ARM microprocessors are used in [77] as the processing unit of the proposed SoC platform for efficient implementation of the Pan & Tompkins ECG QRS detection algorithm [18]. The auto-correlation based ECG peak detection algorithm is optimized by [16] on the DSP processor, and a SoC design is proposed to connect multiple DSP processors for parallel processing of the ECG signals

Table 2.4: Previous biomedical monitoring hardware work

Hard-ware Work	Signal to process	Target Application	Hardware Design Description	Perf. and Energy Effcy.	Flexi-bility*	Multi-Signal Fusion	Fault Tolerance	Comment
Embedded Processors (DSP and ARM)								
Khatib et al. [16]	ECG	Auto-correlation based peak detection	VLIW DSP processors are connected to the system bus to access on-chip and off-chip memories. Up to 12 lead ECG can be analyzed and each lead can be assigned to a DSP for processing.	Med. Low	High	No	No	- DSP and ARM do not offer the best performance and energy efficiency for portable biomedical monitoring, as they are designed for general embedded applications with fine-grained instruction-set level optimizations, such as VLIW instruction issue, circular buffer operations, etc. However, they do not have coarse-grained hardware optimizations specifically designed for the biomedical applications.
Kim et al. [76]	ECG	Wavelet-based peak detection	It is composed of: a) analog front-end to obtain raw ECG signal, and b) DSP back-end to process the collected ECG.	Med. Low	High	No	No	
Chang et al. [77]	ECG	Pan & Tompkins QRS detection	A system-on-chip design architecture is proposed for ECG QRS detection, which incorporates an ARM992T macrocell and other components connected by the system AMBA bus.	Low	High	No	No	
Coarse-Grained Reconfigurable Array (CGRA)								
SYS-CORE [78]	General bio-signals	General biomedical applications	Novel CGRA functional unit and array interconnections are designed for energy-efficient DSP computations, such as FIR filter, matrix multiplication, FFT, etc., that may be useful for the biomedical applications.	Med.	High	No	No	- CGRA provides extra data parallelism and efficiency than DSP and ARM to perform data intensive digital signal processing, but its efficiency depends on the compiler techniques to map the application to the functional unit array. - CGRA configuration is fine-grained and has non-negligible overhead to send the configuration bits. - Many beat detection algorithms are control intensive, instead of data intensive, which CGRA cannot efficiently support.
ULP-SRP [79]	General bio-signals	General biomedical applications	CGRA is used for general biomedical applications by efficiently executing DSP functions. Three execution modes are designed: VLIW and CGRA low/high performance modes, which can be dynamically switched between each other according to the runtime application needs and requirements.	Med.	High	No	No	
(Continued on next page)								

\* Flexibility means how flexible the hardware can be in supporting biomedical applications other than the one it is designed or optimized for.

Table 2.4 (cont.): Previous biomedical monitoring hardware work

Hard-ware Work	Signal to process	Target Application	Hardware Design Description	Perf. and Energy Effcy.	Flexi-bility*	Multi-Signal Fusion	Fault Tole-rance	Comment
FPGA								
Jeong et al. [24] Stojan-ović et al. [25]	ECG	Wavelet-based QRS detection	Specialized hardware blocks are designed and implemented on FPGA for efficient pipelined wavelet decomposition.	Med. High	Low	No	No	- These works just apply FPGA optimizations for the previously developed ECG beat detection algorithms, and have not considered the case when the signal being monitored has low signal quality or the possible algorithm optimizations for further hardware efficiency.
Cvikl and Zemva [80]	ECG	ECG beat detection and classification	An FPGA-based system-on-chip design is implemented, which uses the FPGA logics for ECG beat detection and uses the embedded Power-PC processor on the FPGA board for heart beat classification.	Med. High	Low	No	No	
ASIC								
Pavlatos et al. [23]	ECG	Pan & Tompkins QRS detection	ASIC hardware blocks are designed for each processing stage of Pan & Tompkins ECG QRS detection algorithm, and they are controlled by a central hardwired unit.	High	Low	No	No	- Only the Pan & Tomkins algorithm can run on this proposed ASIC hardware (very low flexibility).
Alemzadeh et al. [62]	ECG ABP HR	Mean, variance, and correlation analysis of the signals	Coarse-grained processing elements are designed, that can be configured to perform mean, variance, and correlation analysis. A majority voter is used to fuse the alarms from different signal analyses.	High	Med. Low	Yes	No	- The applied majority voting scheme is not robust to signal noise. If two signals out of the three are noisy and generate false alarms, even if the third signal is good, false system alarm would be generated, because the applied majority voting does not consider signal qualities.
RR-HMS	ECG ABP	Robust heart rate estimation	Coarse-grained functional units (FUs) are designed, which can be configured to switch between ECG and ABP analysis, as well as for patient-specific monitoring. Weighted sum is used to fuse different signal analyses, where signal qualities are considered.	High	Med.	Yes	Yes	- The efficiency is high, because FUs are coarse grained with application-specific ASIC hardware optimizations. - The flexibility is medium, as it can efficiently support the applications that are able to use the designed FUs. - Hardware fault tolerance is designed with low area and power overheads, by utilizing the application's invariant property.

\* Flexibility means how flexible the hardware can be in supporting biomedical applications other than the one it is designed or optimized for.



from different leads. In addition, Khatib et al. [16] show that DSP has both higher runtime performance and lower energy consumption than ARM when executing the same ECG peak detection algorithm. Similarly, Kim et al. [76] build the SoC design with DSP processors for a wavelet-based ECG peak detection algorithm developed by [81]. DSP and ARM processors are designed and optimized for general DSP and embedded applications with the instruction-level optimizations (such as the VLIW instruction issue, multiply-add operation, circular buffer operation, and others), so they have high flexibility to implement a broad range of applications with relative efficiency. However, as for the specific biomedical monitoring applications or algorithms, DSP and ARM do not offer the best performance and energy efficiency. This is especially important in portable monitoring, which requires higher-level application-specific optimizations beyond the instruction-level optimization.

Coarse-grained reconfigurable array (CGRA) architecture is composed of an array of FUs that execute ALU operations based on the given opcode, such as addition, subtraction, or multiplication. Both the FU operations and array data path are configurable for coarse-grained operations on the data as it flows through the data path. Novel CGRA FU and interconnection designs are proposed in [78] to enable the FU operations and data path interconnections that are useful to efficiently support the DSP operations commonly used in the biomedical signal processing, such as fir filter, matrix multiplication, FFT, and others. In [79], the Samsung reconfigurable processor (with CGRA architecture) is used for low-power biomedical applications, and three operation modes are designed that can be dynamically switched between each other according to the runtime application needs and requirements. The three modes are: CGRA low- and high-performance modes for CGRA operations, and VLIW mode for operations that cannot be mapped to CGRA. The CGRA designs for the biomedical monitoring target general biomedical applications by proposing the hardware architecture for the efficient computations (such as DSP computations) that may be useful in biomedical applications. Compared with DSP and ARM, extra data parallelism and efficiency can be exploited through the function operation and data path configurations in CGRA. CGRA is flexible, but its efficiency highly depends on the compiler techniques to map the application to the array FUs, and frequent configurations are needed in CGRA during the application's ex-

ecution to change the data path for different computations, while CGRA's configuration is fine grained and has non-negligible configuration overheads (both ALU operations and data path need to be configured). In addition, many heart beat detection algorithms [18, 20, 28] are control intensive by applying thresholds, instead of data intensive, which CGRA cannot efficiently support.

To achieve higher performance and energy efficiency for specific biomedical applications, FPGA platforms are applied by [24, 25, 80]. Specialized hardware modules for wavelet-based ECG QRS detection algorithms are built in [24, 25] and implemented on FPGA for the efficient processing of wavelet decomposition and transformation. The ECG beat detection and classification algorithms are partitioned in [80] on the FPGA board, where hardware modules are implemented in FPGA logics for efficient beat detection. Here the embedded PowerPC processor on the FPGA board is used to classify the detected beat as normal or premature ventricular contractions. These FPGA-based systems achieve high efficiency due to the application-specific hardware optimization, but they have low flexibility because the optimized hardware modules are used only for the specific optimized application and cannot be used by other biomedical monitoring applications. In addition, the above FPGA-based systems only use the FPGA capability to optimize the previously developed ECG beat detection algorithms. They do not consider the case when the quality of the single signal being monitored is low, nor do they use algorithm optimizations for hardware efficiency.

The application-specific integrated circuit (ASIC) provides the highest performance and energy efficiency because the specialized hardware modules in ASIC are directly implemented with logic gates, instead of being kept in look-up tables, as in FPGA. Pavlatos et al. [23] implement the Pan & Tompkins ECG QRS detection algorithm with ASIC hardware blocks, where each block is responsible for a different stage of the algorithm, and the blocks are controlled and scheduled by a hardwired ASIC control unit. As a result, Pavlatos et al. [23] achieve very high efficiency but low flexibility. It cannot support any other biomedical applications, even though some of the algorithm stages in other applications could potentially use some ASIC blocks in [23]. In [62], coarse-grained processing elements that can be configured to perform mean, variance, and correlation analysis are designed. The only biomedical monitoring hardware work we found that considers multi-signal analysis and

fusion is [62]. In [62], ECG, ABP, and heart rate (HR) signals are analyzed in the proposed ASIC processing element, and each signal would raise a local alarm if the corresponding analysis detects a problem. The system alarm is raised through majority voting when two or more of the three signals raise local alarms at the same time. However, the applied majority voting mechanism is not always robust to signal noise. If two signals are noisy and cause false local alarms, the false system alarm is generated.

In addition to the work mentioned in Table 2.4, other biomedical monitoring research [82, 83, 84, 85] focuses on signal data acquisition and transmission instead of processing and analysis.

The proposed RRHMS focuses on biomedical signal processing and analysis. It estimates heart rates with multi-signal analysis and fusion. Heart rates are extracted from both ABP and ECG signals, and the final heart rate is obtained by the weighted sum of the two heart rates, where the weights are calculated based on the signal qualities. As a result, RRHMS is more robust to signal noise than the majority voting used in [62] because signal qualities are considered. Coarse-grained ASIC FUs in RRHMS are designed and optimized for high performance and energy efficiency to achieve long battery life. Each FU is responsible for a processing step (stage) of the RRHMS heart rate monitoring algorithm. But unlike the previous work, RRHMS uses algorithmic optimizations to merge the separate ABP and ECG processing steps into shared steps. This allows shared FUs to reduce the hardware area for portable monitoring. Also, RRHMS FUs are controlled and scheduled by a lightweight MIPS processor through the C program with intrinsic functions to invoke FU executions. Therefore, RRHMS is more flexible than the hardwired controller used in [23]. In addition, none of the previous biomedical hardware work has considered hardware fault tolerance. Even though general fault tolerance techniques have been proposed in CGRA and FPGA platforms (introduced in the next section), they usually incur large overheads. In contrast, RRHMS uses its FU design and application invariants to achieve low-overhead fault tolerance.

## 2.4 Hardware Fault Tolerance

Since none of the previous biomedical monitoring hardware works has considered fault tolerance for the processing of biomedical signals, traditional hardware fault tolerance techniques for embedded systems are introduced in this section. These techniques motivate the proposed RRHMS fault tolerance design.

Fault tolerance techniques may exploit redundancy in space (hardware) or in time. Double modular redundancy (DMR) and triple modular redundancy (TMR) are traditional hardware redundancy techniques. They are largely used in embedded hardware designs with ASIC, FPGA, and CGRA [86, 87, 88, 89, 90, 91]. DMR dispatches the same computation to two copies of the hardware and compares the results to detect faults. Similarly, TMR sends computations to three copies of the hardware and compares their results. If the results of the duplicated computations do not match, the fault is detected. In DMR, since it does not know which one of the two executions is faulty, re-execution is needed to recover from the fault [86]. If the fault is transient and does not happen again in re-execution, the two results in re-execution would match and the application proceeds. But if the fault continuously occurs or is permanent, DMR can detect it but not fix it. In TMR, if the fault happens in a single hardware copy while the results of the other two match, the majority voter directly masks the fault without stalling the application (such as by re-execution). If the fault is permanent, TMR can be downgraded into DMR for operations and thus mask the single permanent fault [89].

However, DMR and TMR normally introduce large hardware area and power overheads due to the additional hardware copies and executions. Techniques [89, 90, 91, 92] have been proposed to reduce the hardware area overheads in applying TMR in CGRA and FPGA platforms. However, if the idle resources (spare FUs in CGRA or configuration logic blocks in FPGA) are used, the large power overhead caused by duplicate executions still exists. In addition, the techniques proposed in [93, 94, 95] use the spare resources for data path reconfiguration in FPGA and CGRA to circumvent the logics with permanent faults.

Time redundancy techniques [96, 97, 98, 99, 100] are used to tolerate transient faults while saving overhead in both hardware area and power. They take advantage of the temporal nature of transient faults. Unlike TMR, which

detects and recovers (masks) the fault at the same time, time redundancy techniques usually separate the detection and recovery processes. DMR can be seen as a costly time redundancy technique, as used in [86], where the fault is detected by comparing the results of the DMR processors. On detection of the fault, the two processors are rolled back and re-executed for recovery. This approach is simple in terms of design (just duplicating the hardware modules), but it incurs large hardware area and power overheads. To reduce the overheads in fault detection, [96, 97] proposed a self-checking scheme to distinguish correct and incorrect output values based on the preserved output states. In addition, Jafri et al. [98] use the technique of residue code modulo 3 for low overhead fault detection in the fine-grained arithmetic operations, such as addition and subtraction, by exploiting the modulo 3 relationship between the inputs and outputs. After the fault is detected, time redundancy techniques usually apply re-execution for recovery. Depending on the fault detection delay and hardware implementation, there are three kinds of re-execution [99]:

- (1) *Retry*: the faulty instruction(s) are directly re-executed if the inputs have not been changed or overwritten.
- (2) *Checkpoint and roll-back*: the system state of the previous checkpoint is restored before re-execution (the checkpoint is needed if the input states may be modified before the fault is detected).
- (3) *Restart*: the system needs to be restarted and the whole application re-executed if the fault is detected too late and cannot be recovered through partial application re-execution.

Retry has been applied in [98, 99] to commit the instructions only when no fault is detected during the execution. The checkpoint and roll-back scheme have been applied in [86, 100], and to derive the optimal checkpointing interval, Li and Jiang [86] analytically modeled the system. Restart is used when the fault detection is not effective enough, and it is usually not acceptable in real-time embedded systems, such as the proposed RRHMS.

Therefore, compared with hardware redundancy, such as TMR, time redundancy techniques trade off performance (especially when re-execution is needed) for hardware area and power. We focus on tolerance of transient faults and apply time redundancy for low-overhead fault tolerance design. This is because hardware area and power are crucial for the portability and

battery life of the monitoring system. Additionally, thanks to the RRHMS FU design and optimization, the RRHMS hardware system’s performance easily meets the real-time performance constraint. During normal monitoring without fault, the proposed fault tolerance mechanism does not introduce any performance overhead.

Since traditional fault detection techniques, such as the above introduced ones, try to detect fault in the operations of general applications, they still incur large overheads, though some of them are smaller than the overheads of DMR and TMR. RRHMS exploits application-specific properties to detect faults at low overheads. The following properties of RRHMS are used to achieve this low overhead: (1) coarse-grained FU design, and (2) the application-specific invariant property [26]. Since RRHMS FUs are coarse grained and represent high-level application processing steps, the concept of software invariants proposed in [26] is applicable to the FU outputs. FU output invariants are patient-specific, obtained by application profiling using the patient data. The FU invariants are checked in parallel with normal FU executions and therefore do not introduce performance overhead. An FU finishes the execution only if no fault is detected, otherwise retry is applied to re-execute the FU for transient fault recovery. No extra checkpoint and roll-back schemes are needed in RRHMS, because the inputs of an FU cannot be modified before the completion of its execution (its input states are inherently checkpointed). Similar to the traditional time redundancy techniques, RRHMS only recovers transient faults, even though both transient and permanent faults can be detected.

## Chapter 3

# RRHMS HEART RATE ALGORITHM

In this chapter, the robust heart rate detection algorithm used in RRHMS is introduced. The overview of the algorithm is given first, followed by the detailed explanations of each step in the algorithm, including peak detection, signal quality evaluation, and heart rate estimation and fusion. In the end, the shared processing steps enabled by the algorithmic optimizations are described and discussed.

### 3.1 Algorithm Overview

The robust heart rate is obtained by analyzing both ABP and ECG signals. Algorithmic-level optimizations have been applied to allow shared processing steps between the two signals. Figure 3.1 shows the overall heart rate detection flow, with shared processing steps in grey boxes. There are two main

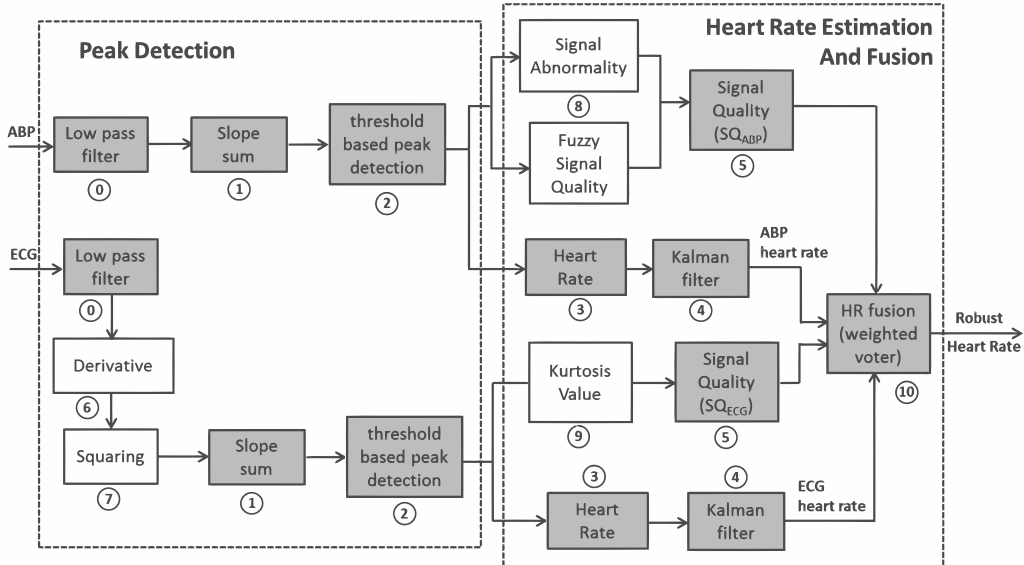


Figure 3.1: Robust heart rate detection flow

stages in the heart rate detection flow:

- (1) peak detection on the raw ABP and ECG signals, which includes signal preprocessing to remove signal noises and highlight approximate peak locations, as well as the threshold-based peak identification, and
- (2) heart rate estimation on each signal and fusion of the two heart rates by a weighted voter (weighted sum) that uses the signal qualities.

Both ABP and ECG signals are collected and analyzed based on 10 s windows. For each 10 s period, the newly collected signals are input to this flow to estimate an average heart rate. This chapter introduces in detail each processing step in this heart rate detection flow.

## 3.2 Peak Detection

Peak detection of both signals starts with a low-pass filter (LPF) to remove high-frequency noises. The LPFs are slightly different for ABP and ECG signals, as they have different natural frequencies in the peaks and surrounding noise interference. We designed the LPFs to be lightweight and easy for computation in an energy efficient portable hardware of the RRHMS, following the fast digital filter design proposed in [8]. The two LPFs for our ABP and ECG signals at 125 Hz sampling frequency are described by the following formulas:

$$\begin{aligned} \text{for ABP:} \quad y_n &= (x_n + 2x_{n-1} + x_{n-2})/4 \\ \text{for ECG:} \quad y_n &= (x_n + 2x_{n-1} + 3x_{n-2} + 2x_{n-3} + x_{n-4})/9 \end{aligned}$$

Since the two LPFs have the same computation structure with different parameters, they can share the hardware computation block with muxes to select parameters.

After the low-pass filter, ECG needs two more steps than ABP does, derivative and squaring. This is because ECG tends to have more low-frequency noise, such as the baseline wonder noise. The derivative is used to differentiate the ECG QRS slope information from the low-frequency noise, and the squaring further emphasizes the higher frequency peaks of ECG. These two steps are part of the ECG preprocessing in the Pan & Tompkins ECG peak detection algorithm [18] to preprocess the ECG signal and highlight its peak



locations.

Next, slope sum is applied on both signals to enhance and smooth the rising portion of the signal for the benefit of the later threshold-based peak detection step. Slope sum was originally used only for the preprocessing of the ABP signal in [20]. However, we found it has similar (sometime even better) effects as the moving-window integration step in the Pan & Tompkins algorithm. Therefore, we applied slope sum for ECG analysis as the replacement for the moving-window integration. Thus, these two originally separate ABP and ECG processing steps are merged into a single step based on their algorithmic purpose. Slope sum is defined in [20] as

$$SSF(k) = \sum_{i=k-w}^k \Delta y_i, \quad b = \begin{cases} \Delta x_i, & \text{if } \Delta x_i > 0 \\ 0, & \text{if } \Delta x_i \leq 0 \end{cases}$$

where  $k$  is the current sample index and  $w$  is the slope sum window ( $w$  should be set as the duration of the signals rising portion). We chose  $w = 15$  (120 ms) for ABP and  $w = 10$  (80 ms) for ECG because ECG has sharper peaks and rises faster than ABP.

After the slope sum, a threshold-based peak detection technique is applied to locate the peaks of the two signals. We developed a peak detection technique that works for both ABP and ECG signals by combining and modifying two previously developed methods for ABP and ECG: threshold-based ABP onset detection [28] and Pan Tompkins threshold-based ECG peak detection [18]. Both of these are among the most popular beat detection methods for ABP and ECG signals, respectively. After the modification, our peak detection method consists of three steps, summarized by the pseudo codes in Algorithm 1. First, detect onset of the slope summed signal by checking a threshold value ( $Th_{onset}$ ) against each slope sum data point. When  $Th_{onset}$  is crossed by a data point, the local search is applied around this point to find the local maximum (slope sum peak) and minimum values. The local searching diameter is half of the estimated peak-to-peak interval ( $T_{est}$ ), centered at the threshold crossing point. Second, check the difference between the local maximum and minimum values, which are found by local searching. If their difference exceeds another threshold ( $Th_{diff}$ ), the slope sum peak is accepted. Then, back searching is applied in the original signal around the slope sum's peak location to detect the peak in the original signal (ABP or

---

**Algorithm 1** Threshold-Based Peak Detection

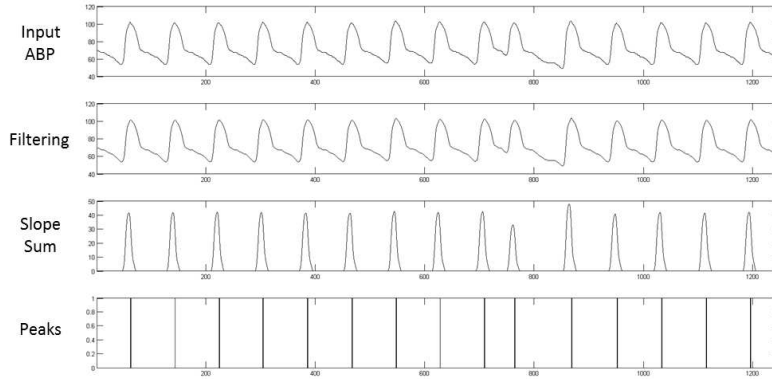
---

```
1: procedure PEAKDETECTION(slope, originSig)
2:    $i \leftarrow 0$ 
3:   while  $i < \text{size of } slope$  (slope summed signal) do
4:     step 1: detect onset of the slope summed signal
5:     if  $slope[i] > th_{onset}$  then
6:       // local search for max. and min. in the slope summed signal
7:        $[slope_{max}, slope_{min}] \leftarrow \text{local search value}(slope, i, \frac{1}{2}T_{est});$ 
8:       // eye closing for half window
9:        $i \leftarrow i + \text{floor}(\frac{1}{2} * T_{est});$ 
10:    step 2: check difference and back search for peak
11:    if  $slope_{max} - slope_{min} > Th_{diff}$  then
12:      // back search in original signal if the peak is accepted
13:       $peak_{idx} \leftarrow \text{local search index}(originSig, i, \frac{1}{4}T_{est});$ 
14:      // eye closing for another half window
15:       $i \leftarrow i + \text{floor}(\frac{1}{2} * T_{est});$ 
16:      step 3: update parameters if the peak accepted
17:      if  $peak_{idx} > \text{last } peak_{idx}$  then
18:        insert peak ( $peak_{idx}$ );
19:        // calculate new parameter values of the detected peak
20:         $new\ th_{onset} \leftarrow \frac{1}{2}slope_{max}$ 
21:         $new\ th_{diff} \leftarrow \frac{1}{2}(slope_{max} - slope_{min})$ 
22:         $new\ T_{est} \leftarrow peak_{idx} - \text{last } peak_{idx}$ 
23:        // update using weighted sum with the above new values
24:         $th_{onset} \leftarrow \frac{7}{8} * th_{onset} + \frac{1}{8} * new\ th_{onset}$ 
25:         $th_{diff} \leftarrow \frac{7}{8} * th_{diff} + \frac{1}{8} * new\ th_{diff}$ 
26:         $T_{est} \leftarrow \frac{7}{8} * T_{est} + \frac{1}{8} * new\ T_{est}$ 
27:       $i \leftarrow i + 1$ 
```

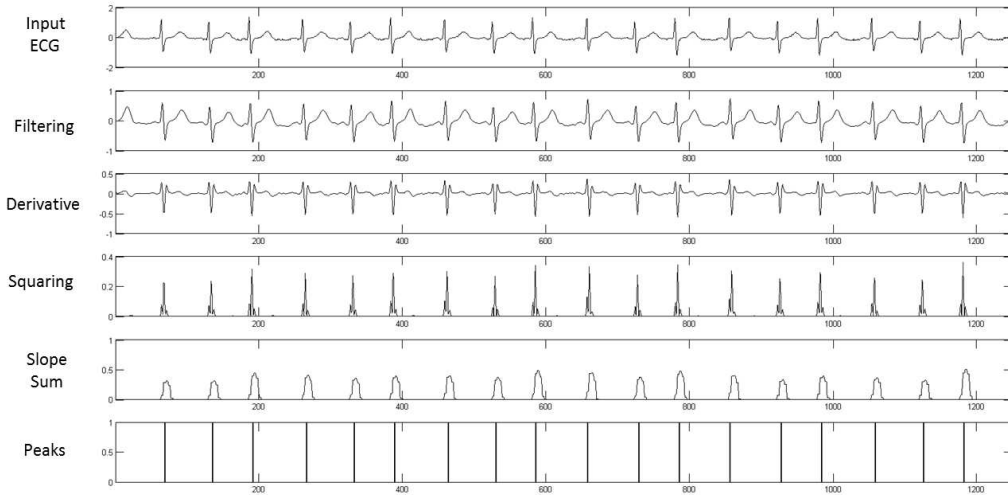
---

ECG) and its location index. The back searching diameter is a quarter of  $T_{est}$ . Back searching has a smaller diameter than local searching for the peaks in the slope summed signal. This is because the peak location of the original signal is close to the corresponding peak location of the slope summed signal. A quarter of  $T_{est}$  is chosen based on experiments with different patient data from the MIMIC II database. At last, check whether the location index of the newly detected peak in the original signal is greater than that of the previously detected peak. This check makes sure that a peak is not detected twice. It is possible that two consecutive peaks may occur within an interval less than a quarter of  $T_{est}$ , such as when the patient starts to have tachycardia. If this check is passed (the newly detected peak is not the dou-

ble detection of an old peak), the new peak is recorded as one of the signal peaks to be output from this peak detection step. Additionally, the three parameters ( $Th_{onset}$ ,  $Th_{diff}$ , and  $T_{est}$ ) are updated with the new parameter values calculated from the newly detected peak, as shown in the pseudo code. This update is used for the dynamic adaption to normal signal changes (e.g., the normal heart rate of the same patient may be different in the morning and in the evening with gradual changes). Weighted sum is applied as the function for updates. Seven-eighths weight is put on the old parameter value and only one-eighth is put on the new value, so that even if the new value is not correct (as when the peak is incorrectly detected from a noisy signal), it does not corrupt the parameters. This is important, because only correct parameters ( $Th_{onset}$ ,  $Th_{diff}$ , and  $T_{est}$ ) enable correct peak detection.



(a) ABP peak detection



(b) ECG peak detection

Figure 3.2: Output of each processing step of peak detection

The initial values of the three parameters ( $Th_{onset}$ ,  $Th_{diff}$ , and  $T_{est}$ ) are trained using the first 20 windows of the patient's data.  $Th_{onset}$  is initialized to be twice the mean slope sum value in the training period, and  $T_{est}$  is the mean peak-to-peak interval.  $Th_{diff}$  is initialized to be half of the average of the maximum and minimum slope sum differences in the entire training period.

Figure 3.2 shows the output of each peak detection processing step for a window of ABP and ECG signals (1250 samples at 125 Hz sampling frequency). From the figure, we can see that all peaks are correctly detected for both ABP and ECG signals.

### 3.3 Signal Quality Evaluation

Signal qualities of the two signals are evaluated and used for the heart rate fusion. The signal qualities are evaluated on the basis of detected beats. ABP signal quality is obtained by combining two previously proposed methods: fuzzy signal quality assessment [20] and signal abnormality assessment [9]. Both methods assign a signal quality value to each detected beat. Fuzzy assessment assigns a fuzzy quality value between 0 and 1 to each beat (the higher the value is, the better quality the beat has). The definition of the fuzzy function is as follow:

$$fuzzy(x, a, b) = \begin{cases} 0, & \text{if } x \leq a \\ 2 * \left(\frac{x-a}{b-a}\right)^2, & \text{if } a < x \leq \frac{a+b}{2} \\ 1 - 2 * \left(\frac{x-a}{b-a}\right)^2, & \text{if } \frac{a+b}{2} < x \leq b \\ 1, & \text{if } b < x \end{cases}$$

Figure 3.3 illustrates the shape of the fuzzy function output. Instead of only output 0 or 1 (binary values), the fuzzy function defines a range ( $[a, b]$ ) where the output is continuous between 0 and 1. Therefore, instead of only classifying a beat as good or bad, the fuzzy function allows the beat quality value to be in between, such as 30% good.

To compute the fuzzy quality, a set of features is extracted from each ABP

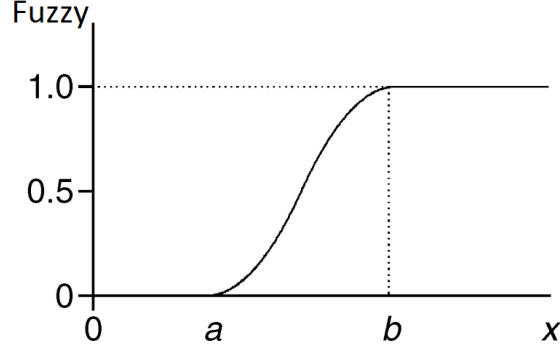


Figure 3.3: Shape of the fuzzy function output [20]

beat: systolic blood pressure (ABP peak value), diastolic blood pressure (ABP valley value), maximum positive blood pressure slope, maximum negative blood pressure slope, maximum up-slope duration, pulse blood pressure (difference between systolic and diastolic blood pressures), and peak-to-peak interval. Each feature value is used as the input (parameter  $x$ ) to a fuzzy function, and the other two fuzzy function inputs (parameter  $a$  and  $b$ ) are either pre-set thresholds or trained base model values. Therefore, a set of fuzzy quality values is computed based on the beat features, and each fuzzy quality computed represents the beat quality in a different aspect. Table 3.1

Table 3.1: Fuzzy functions and their representations of beat quality [20]

Variable		
Name	Representation and Definition	Parameter Explanation
ATL	$ABP\_amplitude\_too\_large:$ $\mu_{ATL} = fuzzy(SBP - SBPa; 20, 60)$	$SBP$ : systolic BP, mmHg; $SBPa$ : systolic BP base
ATS	$ABP\_amplitude\_too\_small:$ $\mu_{ATS} = 1 - fuzzy(DBP; 0, 20)$	$DBP$ : diastolic BP, mmHg
STL	$ABP\_slope\_too\_large:$ $\mu_{STL} = fuzzy(MPPS/MPPSa; 1, 3)$	$MPPS$ : maximum positive BP slope; $MPPSa$ : MPPS base
STS	$ABP\_slope\_too\_small:$ $\mu_{STS} = fuzzy(MNPS/MNPSa; 1, 3)$	$MNPS$ : maximum negative PB slope; $MNPSa$ : MNPS base
KRTL	$ABP\_keeps\_rising\_too\_long$ $\mu_{KRTL} = fuzzy(MUSD; 200, 500)$	$MUSD$ : maximum up-slope duration, ms
SHTL	$ABP\_stays\_high\_too\_long$ $\mu_{SHTL} = fuzzy(MDAT; 400, 800)$	$MDAT$ : maximum duration above threshold, ms
PPD	$ABP\_pulse\_pressure\_decrease$ $\mu_{PPD} = 1 - fuzzy(PBP/PBP a; 0.5, 0.9)$	$PBP$ : pulse blood pressure; $PBP a$ : PBP base
DBPI	$ABP\_diastolic\_pressure\_increase$ $\mu_{DBPI} = fuzzy(DBP/DBPa; 0.8, 1.1)$	$DBP$ : diastolic blood pressure: $DBPa$ : DBP base
PrP	$Premature\_ABP\_pulse:$ $\mu_{PrP} = 1 - fuzzy(T/Ta; 0.75, 0.95)$	$T$ : pulse-pulse interval; $Ta$ : T base

lists all the fuzzy functions we used (similar to Table 1 in [20]), along with their representations for the beat quality. The base model values listed in the parameter explanation column of the table are trained as the average value of the corresponding features. Similar to the parameters in the threshold-based peak detection step ( $Th_{onset}$ ,  $Th_{diff}$ , and  $T_{est}$ ), the first 20 windows of patient data are used to train the base model values as well, and the base model values are updated with the weighted sum function:

$$V = 0.875 * V + 0.125 * V_{new}$$

where  $V$  on the right of the equal sign is the previous base model value and  $V$  on the left of the equal sign is the updated base model value.  $V_{new}$  is the feature value extracted from the current beat. The base model values are updated only if the beat is not classified as abnormal by the signal abnormality assessment (introduced later).

With all of the fuzzy values (in Table 3.1) computed from each beat feature, three composite quality values are defined by using fuzzy conditional statements: *ABP\_amplitude\_normal*, *ABP\_slope\_normal*, and *ABP\_with\_blocked\_transducer*. The composite quality values are the intermediate values used to derive the final beat quality. The following definitions to compute the composite values and final fuzzy quality value are from [20]. We used the same method as in [20] to derive the final fuzzy quality for the beat in RRHMS. *ABP\_amplitude\_normal* is defined as:

$$\begin{aligned} \text{IF} \quad & [\text{not } ABP\_amplitude\_too\_large \text{ (ATL)}] \quad \text{and} \\ & [\text{not } ABP\_amplitude\_too\_small \text{ (ATS)}] \\ \text{THEN} \quad & ABP\_amplitude\_normal \text{ (AN)} \\ & \mu_{AN} = (1 - \mu_{ATL}) \wedge (1 - \mu_{ATS}) \end{aligned}$$

where  $\wedge$  stands for fuzzy intersection and is defined as  $\mu_A \wedge \mu_B = \min[\mu_A, \mu_B]$ . Similarly, the other two composite quality values are computed as follows:

$$\begin{aligned} \text{IF} \quad & [\text{not } ABP\_slope\_too\_large \text{ (STL)}] \quad \text{and} \\ & [\text{not } ABP\_slope\_too\_small \text{ (STS)}] \\ \text{THEN} \quad & ABP\_slope\_normal \text{ (SN)} \\ & \mu_{SN} = (1 - \mu_{STL}) \wedge (1 - \mu_{STS}) \end{aligned}$$

**IF**        [ *ABP\_pulse\_pressure\_decrease* (PPD)]                    **and**  
               [ *ABP\_diastolic\_pressure\_increase* (DBPI)]                **and**  
               [ **not** *premature\_ABP\_pulse* (PrP)]  
**THEN**    *ABP\_with\_blocked\_transducer* (WBT)  

$$\mu_{WBT} = \mu_{PPD} \wedge \mu_{DBPI} \wedge (1 - \mu_{PrP})$$

In the end, the final fuzzy quality of the beat is defined as:

**IF**        [ *ABP\_amplitude\_normal* (AN)]                                **and**  
               [ *ABP\_slope\_normal* (SN)]                                    **and**  
               [ **not** *ABP\_keeps\_rising\_too\_long* (KRTL)]                **and**  
               [ **not** *ABP\_stays\_high\_too\_long* (SHTL)]                **and**  
               [ **not** *ABP\_with\_blocked\_transducer* (WBT)]  
**THEN**    *ABP\_signal\_quality\_good* (SQG)  

$$\mu_{SQG} = \mu_{AN} \wedge \mu_{SN} \wedge (1 - \mu_{KRTL})$$

$$\wedge (1 - \mu_{SHTL}) \wedge (1 - \mu_{WBT})$$

On the other hand, the signal abnormality assessment of ABP gives a binary value, 0 or 1, to each beat (0 means normal beat, 1 means abnormal beat). The beat is classified as normal or abnormal by checking the features extracted from the beat with the pre-set thresholds. Table 3.2 lists the abnormality criteria used in [9]. The listed thresholds are chosen to be the

Table 3.2: Criteria for abnormal beat detection [9]

Feature	Description	Abnormality Criteria
<i>SBP</i>	systolic blood pressure	$SBP > 300$ mmHg
<i>DBP</i>	diastolic pressure	$DBP < 20$ mmHg
<i>MBP</i>	mean blood pressure	$MBP < 30$ or $MBP > 200$ mmHg
<i>HR</i>	heart rate	$HR < 20$ or $HR > 200$ bpm
<i>PBP</i>	pulse blood pressure	$PBP < 20$ mmHg
<i>w</i>	mean negative slopes	$w < -40$ mmHg/100 ms
$SBP[k] - SBP[k-1]$	systolic BP difference	$ \Delta SBP  > 20$ mmHg
$DBP[k] - DBP[k-1]$	diastolic BP difference	$ \Delta DBP  > 20$ mmHg
$T[k] - T[k-1]$	peak interval difference	$ \Delta T  > 2/3$ s

physiological limits of human beings. For example, the systolic blood pressure never exceeds 300 mmHg and diastolic blood pressure never drops below 20 mmHg. So if any criterion is violated, it means the detected beat is abnormal. We modified the abnormality criteria in RRHMS. Instead of using the pre-set thresholds, we use the trained base models for each of the features listed in Table 3.2 except the last three difference features. Table 3.3 shows the modified criteria used in RRHMS, where  $X_a$  means the base model value of feature  $X$  (similar to the notation used in Table 3.1). Since most of the features used in abnormality detection are also used in the fuzzy signal quality assessment discussed above, the training and updating of the base models for abnormality detection can share the computations in the fuzzy quality assessment and therefore not add much computation complexity. We replace each upper limit threshold with twice the corresponding trained base model value and replace the lower limit threshold with half of the corresponding trained base model value, making our modified criteria are more patient-specific. The criteria for the last three difference features are not modified, because they are very sensitive to the data in the training period. If during the training period, the patient's ABP waveform does not change much, then the difference features are close to 0. As a result, during the monitoring, a small change in the ABP waveform (even within normal physiological range) may cause violations in the criteria of the difference features. So we used the same pre-set thresholds as in [9] for those three difference features. The other features do not have this problem because they are not affected by the

Table 3.3: Modified criteria for abnormal beat detection used in RRHMS

Feature	Description	Abnormality Criteria
$SBP$	systolic blood pressure	$SBP > 2*SBP_a$
$DBP$	diastolic pressure	$DBP < 0.5*DBP_a$
$MBP$	mean blood pressure	$MBP < 0.5*MBP_a$ or $MBP > 2*MBP_a$
$HR$	heart rate	$HR < 0.5*HR_a$ or $HR > 2*HR_a$
$PBP$	pulse blood pressure	$PBP < 0.5*PBP_a$
$w$	mean negative slopes	$w < 2*w_a$
$SBP[k] - SBP[k-1]$	systolic BP difference	$ \Delta SBP  > 20 \text{ mmHg}$
$DBP[k] - DBP[k-1]$	diastolic BP difference	$ \Delta DBP  > 20 \text{ mmHg}$
$T[k] - T[k-1]$	peak interval difference	$ \Delta T  > 2/3 \text{ s}$



changes in the training period. For example, if the average systolic blood pressure of a patient in the training period is 120 mmHg, a newly detected systolic blood pressure of more than 240 mmHg or less than 60 mmHg is always an indicator of abnormality. As mentioned above, all the base model values are updated only if the beat is classified as normal (i.e., if no criteria are violated).

From the two signal quality values (fuzzy signal quality and signal abnormality), the overall quality of each beat is calculated as follows: if the beat is evaluated as normal (abnormality value is 0), the fuzzy quality value is directly used as the beat quality. Otherwise, the fuzzy quality value multiplied by 0.7 is used as the beat quality. At last, the overall ABP signal quality for the window,  $SQ_{ABP}$ , is judged by the qualities of all the detected beats within this window. If there is no beat detected in the window, it means either the patient is having the asystole problem or the sensor is disconnected (sensor disconnection is not a rare problem in the MIMIC II database). For either case, we assign the window signal quality to 0. If both ABP and ECG window signal qualities are zero for either reason, a flag will be raised in the heart rate fusion step that fuses the heart rate based on both signal qualities. This is the best we can do because currently we have no way to differentiate asystole and sensor disconnection. The ABP signal quality of the window is defined as the percentage of the detected beats whose quality is beyond a threshold ( $SQ_{TH}$ ):

$$SQ_{ABP} = \frac{1}{N} \sum_{i=0}^{N-1} \mathbf{1}_{\{x_i > SQ_{TH}\}}$$

where  $\mathbf{1}_{\{cond\}}$  is an indicator function, which outputs 1 if  $cond$  is evaluated to be true, otherwise 0.  $N$  is the total number of beats detected in this window, and  $x_i$  is the final fuzzy quality of the  $i$ th beat.  $SQ_{TH}$  is set to be 0.5 in our implementation, which is also the value used in [20] to indicate good quality ABP beat.

Similarly, the kurtosis value is computed for each detected ECG beat to evaluate the beat quality. Kurtosis is the fourth standardized moment, defined by the formula

$$\text{kurtosis} = \frac{E[(X - \mu)^4]}{(E[(X - \mu)^2])^2} = \frac{\mu^4}{\sigma^4}$$

where  $\mu$  is the mean value of the array (raw ECG data values) and  $\sigma$  is the standard deviation. The kurtosis value of an ECG beat is computed using two periods around the current beat (from the previous detected beat to the next detected beat). Low kurtosis value indicates low-frequency noise in the ECG signal [70]. Therefore, the final ECG signal quality of the window,  $SQ_{ECG}$ , is defined as the percentage of the beats whose kurtosis value is above a threshold. The same formula used to compute the final ABP signal quality can be also used for computation of the final ECG signal quality, where  $x_i$  in this case stands for the kurtosis value of the  $i$ th ECG beat, and  $SQ_{TH}$  is set to 5 to differentiate the good and noisy ECG beats, according to [70].

### 3.4 Heart Rate Estimation and Fusion

After peak detection, the average heart rate is calculated from each signal using the formula

$$HR = \frac{60}{N} \sum_{i=1}^N T_i \quad (\text{beats/min})$$

where  $T_i$  is the peak to peak interval (in seconds) between the  $i$ th and  $(i-1)$ th peak. Then, the Kalman filter is applied to remove high-frequency noise in the heart rate estimations between the signal windows [21].

In the end, the Kalman-filtered heart rates from the two signals ( $HR_{ABP}$  and  $HR_{ECG}$ ) are weighted to obtain the final heart rate estimation for this window. Weights are calculated based on four parameters: ABP and ECG signal qualities ( $SQ_{ABP}$  and  $SQ_{ECG}$ ) and their Kalman residuals obtained from the Kalman filtering process ( $r_{ABP}$  and  $r_{ECG}$ ). Similar to [21], the weights and final weighted heart rate are calculated by

$$\text{weighted HR} = \frac{\omega_2}{\omega_1 + \omega_2} * HR_{ABP} + \frac{\omega_1}{\omega_1 + \omega_2} * HR_{ECG}$$

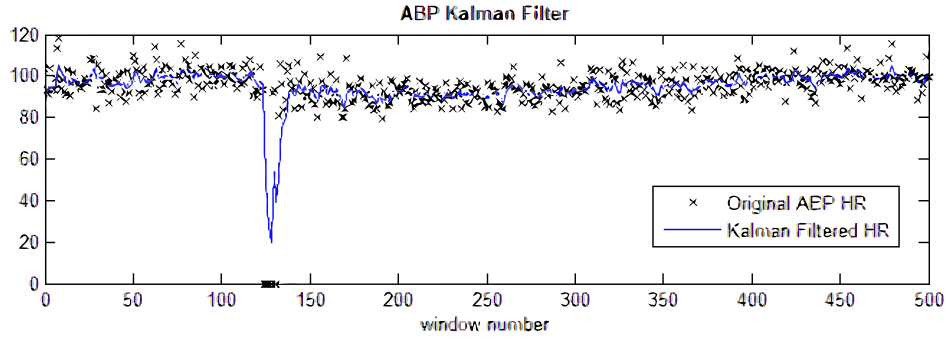
$$\text{where,} \quad \omega_1 = \left(\frac{r_{ABP}}{SQ_{ABP}}\right)^2 \quad \text{and} \quad \omega_2 = \left(\frac{r_{ECG}}{SQ_{ECG}}\right)^2$$

Interested readers can refer to [21] for more details on weighted heart rate estimation using the Kalman filter and signal qualities. Figure 3.4 and Figure 3.5 show two examples of the Kalman filtered heart rate and final weighted

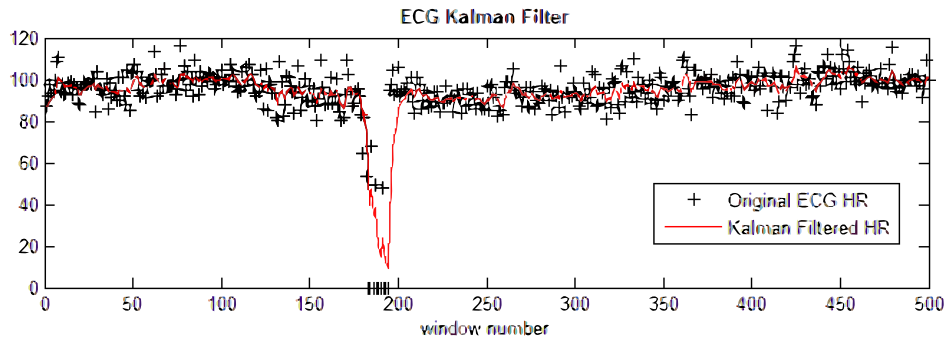
heart rate for 500 windows of ABP and ECG data, collected from the MIMIC II database (patient number a41709 and a41178, respectively). In both examples, we see that the Kalman filter is able to remove high-frequency noise and smooth the heart rate estimations between windows. In Figure 3.4, at around window 128, the ABP signal is corrupted by large artifacts and noise, so no peaks are detected and the signal quality is low. Similarly, the ECG signal is corrupted around window 187. These two segments are cases where one signal is corrupted while the other signal is good. The same scenario happens in Figure 3.5 at around window 50 as well, where ECG is corrupted while ABP is good. In the end, the weighted sum voter fixes these problems, by weighting less on the low-quality signals at the corresponding segments, and therefore it enables accurate and continuous heart rate monitoring.

### 3.5 Shared Processing

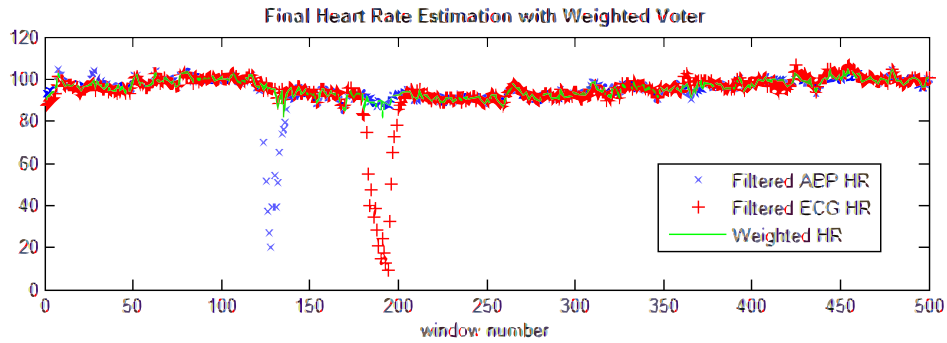
As highlighted in the gray boxes of Figure 3.1, most of the processing steps are shared for ABP and ECG analyses. The shared processing steps have the same computations but with different parameters for ABP and ECG signals (e.g.,  $w$  value in slope sum,  $Th_{onset}$ ,  $Th_{diff}$ ,  $T_{est}$  values in peak detection, etc.). In addition, some parameters are different for different patients, such as the base model values in the ABP signal quality assessment. The sharing allows shared hardware modules for performance- and energy-efficient computation within a tight area constraint. The signal- and patient-specific parameters can be passed to the hardware modules through configuration registers in the module. Therefore, within only a few cycles, a module is able to be configured to switch between ABP and ECG computations, as well as between different patients. The fast configuration reduces the overhead of shared hardware modules for the computations of different biomedical signals and enables fast overall processing speed, which allows frequency scaling to reduce the average power consumption. More details of the hardware design are introduced in the following section (Chapter 4).



(a) Kalman filtered ABP heart rates

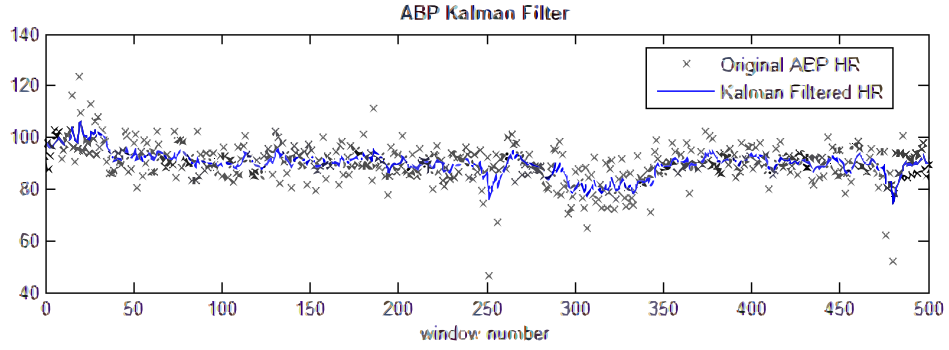


(b) Kalman filtered ECG rates

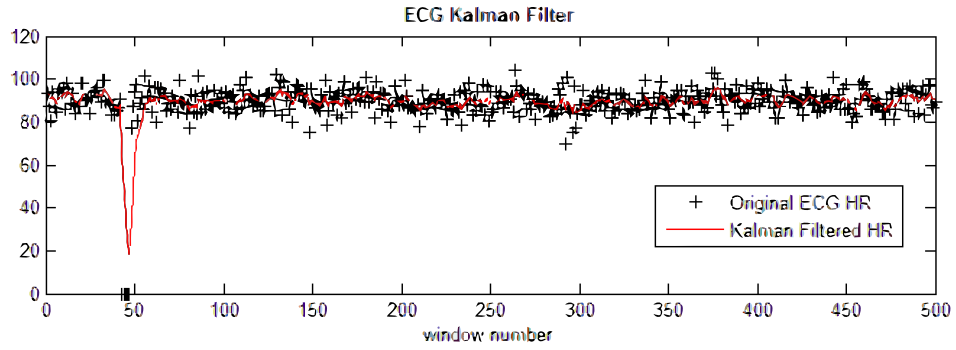


(c) Weighted heart rate

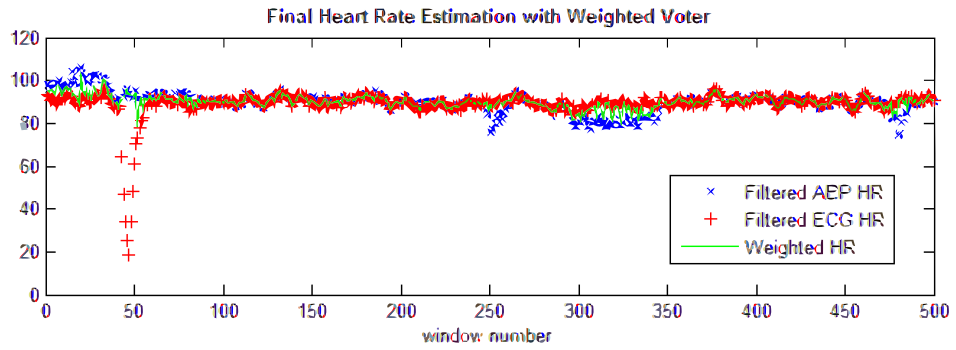
Figure 3.4: Kalman filtered ABP and ECG heart rates for 500 windows, and final weighted heart rates of the corresponding windows (data from the MIMIC II database patient a41709)



(a) Kalman filtered ABP heart rates



(b) Kalman filtered ECG rates



(c) Weighted heart rate

Figure 3.5: Kalman filtered ABP and ECG heart rates for 500 windows, and final weighted heart rates of the corresponding windows (data from the MIMIC II database patient a41178)

# Chapter 4

## RRHMS HARDWARE SYSTEM

This chapter introduces the baseline design of the RRHMS hardware system without fault tolerance features (fault tolerance is introduced in Chapter 5). It begins with an overview of the hardware architecture, followed by the detailed description of the configurable FU design. The FUs are configured and controlled by a central MIPS controller, which is discussed after FU design. We conclude by explaining the mapping and implementation of the robust heart rate estimation algorithm (introduced in Chapter 3) for the proposed hardware system.

### 4.1 Hardware System Overview

The proposed hardware system consists of three main parts, as shown in Figure 4.1: (1) an FU ASIC accelerator composed of a set of configurable FUs, (2) a lightweight MIPS controller, and (3) a shared on-chip memory system.

The FUs efficiently execute the processing steps of the robust heart rate estimation algorithm (processing steps are summarized in Figure 3.1). Therefore, the FUs are coarse grained, and ASIC optimizations can be applied to each FU at a large computation scope to achieve high performance and energy efficiency. As mentioned in Section 3.5, the FUs are designed by taking advantage of the algorithmic optimizations, where the separate processing steps of ABP and ECG signals are shared. As a result, most FUs are responsible for both ABP and ECG computations. Different parameters used in ABP and ECG computations are passed to the FUs through the configuration registers inside them.

The FUs are controlled by a lightweight MIPS controller, which is modified from a 16-bit open source MIPS processor downloaded from the OpenCores

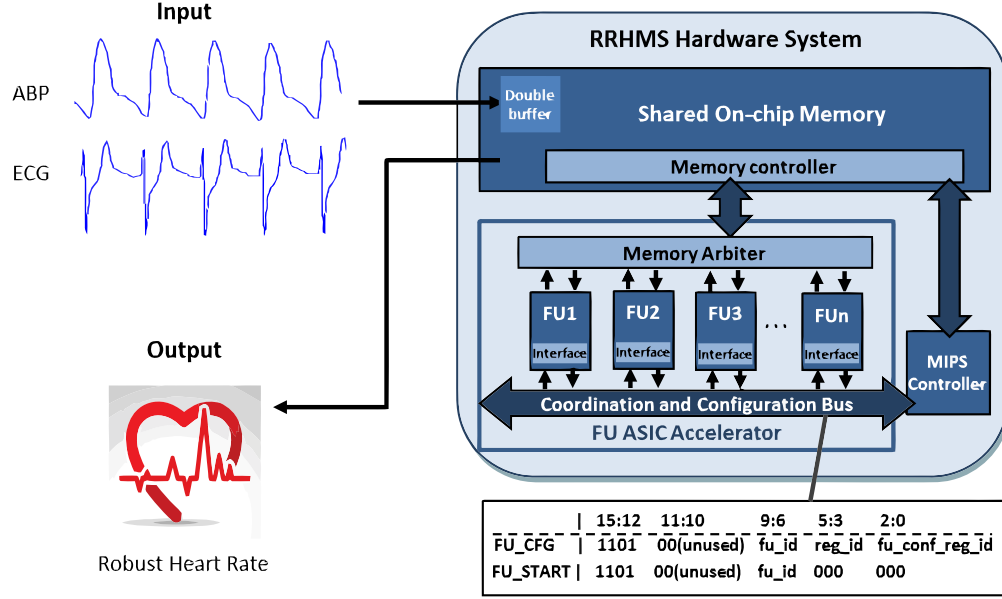


Figure 4.1: RRHMS hardware system overview

Community [101]. Two instructions are extended from the MIPS baseline instruction set to configure and execute the FUs. The two instructions are sent from the MIPS controller to the corresponding FUs through the system coordination and configuration bus, as illustrated by Figure 4.1 (lower right part). The MIPS controller is also responsible for other computations that are not supported by the coarse-grained FUs, such as control flows.

The shared on-chip memory is used for three purposes: (1) general memory for FUs and MIPS controller to store and load computation data, (2) communication channel for FUs and MIPS controller to pass data and computation results to each other, and (3) interface to collect the input biomedical signals and output the computed heart rate. To support real-time processing, the input signals are stored in a double buffer. When a window of signal data is ready, the MIPS controller is notified to start the processing of this window. At the same time, the next window of incoming signal data is stored in the other half of the double buffer through direct memory access. In this way, processing is not interrupted by the incoming data, and if the hardware fault is detected, the current window can be re-processed for transient fault recovery (discussed in Chapter 5), since the current window’s data have not been overwritten by new data.

All FUs are designed following the same template and have the same in-

interfaces (connected to system bus and memory). The only difference is the computations they support. Therefore, it is easy to remove and add FU components to modify the proposed hardware system for other embedded applications. In this sense, not only can the proposed hardware system be used in RRHMS to efficiently run the robust heart rate detection application, it can also be used as a framework for embedded hardware designs.

In RRHMS, the proposed hardware system runs the robust heart rate detection algorithm discussed in Chapter 3. As depicted in Figure 4.1 (left part), the inputs to the hardware system are raw ABP and ECG signals collected from the bio-sensors, and they are stored in the dedicated memory locations (double buffer). The output is the weighted heart rate estimation obtained from the analysis of the ABP and ECG signals. The output is also stored in the dedicated memory location so that other systems can read it from the RRHMS to make sure of the heart rate information, such as the heart rate variability analysis system.

## 4.2 Functional Unit Design and Configuration

Functional units (FUs) are a set of coarse-grained accelerators to support efficient ABP and ECG processing. Each processing step shown in Figure 3.1 has a corresponding FU implementation. With the computation sharing enabled by the algorithmic optimization, there are a total of 11 FUs needed for our heart rate detection algorithm. (FU numbers are listed next to the corresponding processing steps in Figure 3.1). All FUs are designed to have the same interface and follow the same design template as depicted in Figure 4.2. Each FU is composed of three parts: (1) interfaces (bus and memory interface), (2) configuration registers (CRs), and (3) computation logics (data path and state machine controller).

The FU bus interface connects to the system coordination and configuration bus (between MIPS controller and ASIC accelerator). It monitors the configuration and execution instructions sent from the MIPS controller. Upon receiving a configuration instruction corresponding to its FU, the bus interface reads the instruction from the bus, parses the configuration parameters, and configures the CR in its FU as specified by the instruction. When the execution instruction is received, the bus interface notifies the FU com-



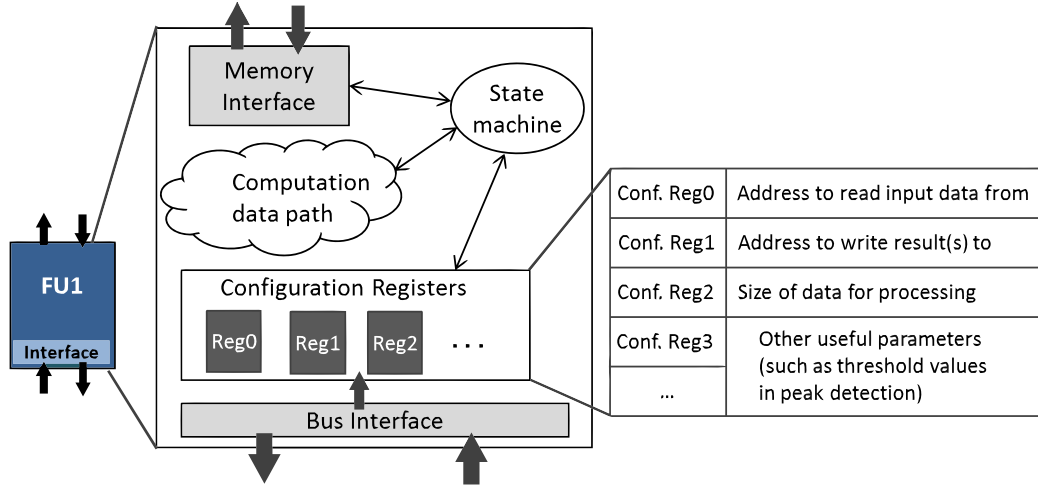


Figure 4.2: Template of functional unit design

putation logics to start execution. When the computation logics finish the FU execution, the bus interface sends a DONE signal with its FU number to the coordination and configuration bus to let the MIPS controller know about its completion. FU memory interface, on the other side, is responsible for reading and writing data from and to the on-chip memory shared between the MIPS controller and FU ASIC accelerator. All FU memory interfaces connect to the memory arbiter, which is a priority selector used to schedule memory requests from all FUs. FU priorities can be set by users to overwrite the default round robin priority setting. Similarly, FU priorities are also used on the bus interface side to resolve contention in the DONE signals when multiple FUs finish executions at the same time, as the lightweight MIPS controller can handle only one DONE signal in a cycle. There is also a priority selector for FU DONE signals, but its complexity is much less than that of the memory arbiter. When the request of an FU, either to memory arbiter or to DONE arbiter, is granted, the FU is notified to continue its execution. Otherwise, the FU keeps sending the request and waiting for its feedback. The priority and arbiter features have been implemented in the current design but are not in use because currently the FUs are scheduled to be executed one by one, not in parallel. For these reasons, there is no contention for either memory requests or DONE signals. However, pipelining the FU executions is planned for future work. With pipelined execution, multiple FUs may request memories or finish executions in the same cycle, but the priority scheme will effectively resolve these contentions. An FU

whose result is waited for by other FUs should be set with higher priority. Therefore, the FU interfaces are designed for the scalability of the system, and more FUs can be easily added.

FU configuration registers (CRs) function as the bridge for passing the needed input parameters into the FU computation logics. The most common parameters used by FUs (listed in Figure 4.2) are: memory address to read input data, memory address to store computation result(s), and size of the input data to process. These parameters provide the FU with the basic information about the input data and the way to output the result. Additionally, more CRs can be added in an FU for passing other useful parameters, such as the slope sum window size ( $w$ ) in slope sum computation, threshold and estimated period values ( $Th_{onset}$ ,  $Th_{diff}$ ,  $T_{est}$ ) in peak detection, base model values in ABP signal quality assessment, and others. Therefore, each FU may have a different number of CRs (up to eight are allowed in our implementation). If the FU needs more input parameters than are allowed, some inputs can be grouped into a consecutive memory chunk. A CR can then pass the starting address of the memory chunk to the FU for loading and computation. Table 4.1 shows all the configuration registers in each of the 11 FUs implemented in the RRHMS.

FU computation logics are made up of two parts: (1) computation data path, and (2) state machine controller. The computation data path is FU-specific and implements the ASIC logic of the corresponding FU computation. The state machine is the hardwired controller that

- schedules the computation in the data path,
- reads configuration registers for input parameters,
- sends memory requests through the memory interface, and
- listens to the bus interface for execution start signal and notifies the bus interface to send the DONE signal when the execution completes.

The state machine controllers (SMCs) are similar in all FUs and follow the same scheduling pattern. At the beginning, the SMC is in the *IDLE* state. When an execution signal is sent from the MIPS controller corresponding to its FU, the SMC is notified by the bus interface to start execution. In the execution, SMC first switches to *READMEM* state, where it checks the corresponding configuration register for the input data address and notifies the memory interface to send the memory request to read the first input

Table 4.1: Functional units and their configuration registers (CR)

FU No.	FU Name	FU Description	Configuration Registers
0	low-pass filter	Remove high-frequency noise in ABP and ECG signals.	CR0: address to read raw ABP/ECG signal CR1: address to store filtered signal CR2: signal size, last bit (0 - ECG or 1 - ABP)
1	slope sum	Compute the slope sum values according to the slope sum function (Section 3.2).	CR0: address to read the signal CR1: address to store slope sum result CR2: signal size CR3: slope sum window size
2	peak detection	Detect the peak locations (or indexes) for the input signal (ABP or ECG) of the current window.	CR0: address to read raw ABP/ECG signal CR1: address to store detected peak indexes CR2: signal size CR3: address to read the slope summed signal CR4: address of model parameters ( $Th_{onset}, Th_{diff}, T_{est}$ ) CR5: address to store detected peak number CR6: address to store detected valley indexes
3	heart rate	Calculate the average heart rate of the current window.	CR0: address to read peak locations CR1: address to store computed heart rate CR2: number of peaks
4	Kalman filter	Filter the high-frequency noise in the heart rate estimations between windows.	CR0: address to store filtered (estimated) heart rate CR1: address of input parameters (posteriori value, measured heart rate, last estimated heart rate) CR2: address to store computed Kalman residue
5	final signal quality	Calculate the percentage of high-quality beats in the current window.	CR0: address to read signal quality of each beat CR1: address to store signal quality (percentage) result CR2: number of beats CR3: threshold for high quality beat (upper 16 bits) CR4: threshold for high quality beat (lower 16 bits)
6	derivative	Compute the derivative value of the input signal.	CR0: address to read the input signal CR1: address to store the computed derivative values CR2: signal size
7	squaring	Compute the squaring value of the input signal.	CR0: address to read the input signal CR1: address to store the computed squaring values CR2: signal size
8	ABP beat quality	Evaluate the signal quality for a detected ABP beat (combining techniques developed in [9] and [20]).	CR0: address to read ABP signal CR1: address to store the computed beat quality CR2: peak index for the current evaluated beat CR3: last peak index CR4: valley index for the current beat CR5: next valley index CR6: address to read base model parameters
9	ECG beat quality	Evaluate the signal quality for a detected ECG beat (kurtosis value).	CR0: address to read ECG signal (from the previous beat peak to next beat peak) CR1: address to store the beat quality result (computed kurtosis value) CR2: signal size from the previous to next beat peak
10	heart rate (HR) fusion	Fuse the heart rates computed from ABP and ECG signals (by weighted sum).	CR0: address to load the input parameters (signal qualities, Kalman residues, and heart rates of both ABP and ECG signals) CR1: address to store the fused heart rate

Note: address in the table means memory address, and CR means configuration register (16 bit).

data. The input data come in after a cycle from the on-chip memory (Block RAM in FPGA implementation or SRAM in ASIC implementation). After getting the first input data, SMC jumps to the *READMEM\_AND\_COMP* state to continue sending memory requests to read the following input data. At the same time, it notifies the computation logics to process the new input data that come in due to the memory request previously sent. Therefore, the computations are pipelined with the memory reads. If in a cycle, no input data come in due to the scheduling of the memory arbiter, no notification is sent to the computation logics to process the input data, and the same memory request is sent again. After all the input data have been read, SMC enters *WAIT\_COMP* state to wait for the computation logics to finish the computation. During this time, the SMC may still send different control bits to computation logics according to the computation stage. When the computation is finished and the result(s) have been written to the memory through the memory interface, SMC goes to *WAIT\_DONE* state. In this state, it lets the bus interface keep sending the DONE signal until its DONE signal is processed by the MIPS controller. After this, the SMC jumps back to *IDLE* state and waits for the next round of execution. This is the general SMC scheduling pattern followed by all the FU SMCs, and all FU computation logics are pipelined and optimized in all FU implementations as well. The only difference between the FUs is the specific data path computation and number of data path stages.

Since all FUs have the same interface and follow the same design template, the proposed hardware system can be used as a framework that can be tailored (by adding or removing FU modules) to support other embedded applications.

### 4.3 MIPS Controller

The MIPS controller is lightweight. It occupies about only 5% of the processing area (total area except memory: FU ASIC accelerator + MIPS controller). We designed the MIPS controller in the RRHMS by making the following modifications on the 16-bit open source MIPS processor downloaded from the OpenCores Community [101]:

- Changed the data path from 16-bit to 32-bit to allow higher precision

of computations (instruction width is kept as 16 bit).

- Added forwarding logic to resolve the read-after-write (RAW) hazard in fewer cycles compared with stalling the pipeline in the original design.
- Extended the baseline instruction set with two more instructions to configure and execute FUs (by adding logics in the ID and MEM pipeline stages).

The MIPS controller is responsible for: (1) configuring the FUs by sending them configuration parameters, (2) scheduling the execution of FUs by sending them execution instructions, and (3) executing the basic MIPS instructions needed between FU executions, such as the control flows and glue-logic computations.

The FU configuration and execution instructions sent from the MIPS controller are realized by extending the MIPS instruction set. Two instructions are added to the base instruction set, as shown in Figure 4.1 (lower right part):

- (1) FU configuration (**FU\_CFG**) instruction which moves a configuration parameter from a MIPS register (*reg\_id*) to a configuration register (*fu\_conf\_reg\_id*) of an FU (*fu\_id*), and
- (2) FU execution (**FU\_START**) instruction which notifies an FU (specified by *fu\_id* field) to start execution.

Once an FU finishes the execution and stores the results to the shared memory, the MIPS controller is notified by the FU's bus interface, which sends a DONE signal to the system coordination and configuration bus.

To illustrate how the two extended instructions work, Figure 4.3a shows an example C code running on the MIPS controller. This simple example shows how the FUs are executed from the C code. In the example, the C code first reads a window of signal data into the *signal* array (1250 samples for a 10-second window of the signal sampled at 125 Hz) and then computes the slope sum value of the *signal* array and stores the slope sum result into the *slope\_result* array. The FU executions are invoked in the C code by intrinsic functions recognizable by the compiler. An extended version of the MIPS C compiler can be used to generate the assembly code from C programs. To map the C intrinsic function to the corresponding FU, the compiler needs to maintain a table of this mapping and know the meaning of

```

#define size 1250

int main()
{
    int signal [size];
    read_signal (signal, size);
    ....
    int slope_result [size];
    Slope_Sum (slope_result , signal, size, w);
    ....

    return 0;
}

```

(a) Example C code running in the MIPS controller

```

Slope_Sum(int result[], int signal[], int size, int w):
    // configure starting address to read input
1. mov    #signal, reg1 // reg1 = signal address
2. fu_cfg fu1, reg1, fu_conf_reg_0 // cr0 = reg1 = signal addr
    // configure starting address to store result
3. mov    #result, reg1 // reg1 = result addr
4. fu_cfg fu1, reg1, fu_conf_reg_1 // cr1 = reg1 = result addr
    // configure the size of signal array
5. load    size, reg1 // reg1 = size
6. fu_cfg fu1, reg1, fu_conf_reg_2 // cr2 = reg1 = size
    // configure slope sum parameter w
7. load    w, reg1 // reg1 = w
8. fu_cfg fu1, reg1, fu_conf_reg_3 // cr3 = reg 1 = w
    // start execution of slope sum computation on FU1
9. fu_start fu1
    // after execution, result array stores the slope sum values
10. ret

```

(b) Assembly code of the compiled Slope\_Sum intrinsic function that runs on FU1

Figure 4.3: Example code running in MIPS controller

the FU configuration registers. In the example, **Slope\_Sum** is an intrinsic function, and by looking up the intrinsic function to the FU mapping table, the compiler finds it is mapped to FU1, which is the FU responsible for slope sum computation. As a result, the assembly code shown in Figure 4.3b is

generated by the compiler for the **Slope.Sum** function. In lines 1 to 8 of Figure 4.3b, FU1 is configured through the FU\_CFG instructions. In line 9, the execution is started by the FU\_START instruction. During the execution, FU1

- (1) reads the data of the *signal* array from the memory, whose address is specified by the configuration register 0 (CR0) of FU1,
- (2) computes the slope sum values with the configured slope sum window size specified by CR3, and
- (3) writes the computed slope sum result to the *result* array, whose memory address is specified by CR1.

The above execution steps are pipelined inside FU1, and the total size of the *signal* array data to read and process is specified by CR2.

## 4.4 Robust Heart Rate Application Mapping

In the initial prototype implementation, FUs are implemented for each processing step of the robust heart rate detection algorithm (Figure 3.1). So the application mapping on the hardware system is supposed to be straightforward, done simply by calling the intrinsic functions following the order of the heart rate detection flow and passing the corresponding parameters to the FU configuration registers. But since we do not have the modified MIPS compiler yet to generate the assembly code from the C program, we manually wrote the assembly code to configure and execute the FUs for heart rate detection. We modified the assembler that comes with the open source MIPS processor to support the extended instructions and used it to assemble the manually written assembly code. After the assembling process, the assembled machine code is put into the instruction memory of the hardware system. Then the heart rate detection application starts to run after resetting the program counter (PC) to point to the first machine code instruction.

In the application, the FUs are first configured for ABP processing to extract all the ABP features used for heart rate fusion: ABP Kalman filtered heart rate ( $HR_{ABP}$ ), ABP Kalman residue ( $r_{ABP}$ ), and ABP window signal quality ( $SQ_{ABP}$ ). Then the FUs are configured with ECG parameters for ECG processing to extract the corresponding ECG features: ( $HR_{ECG}$ ,  $r_{ECG}$ ,

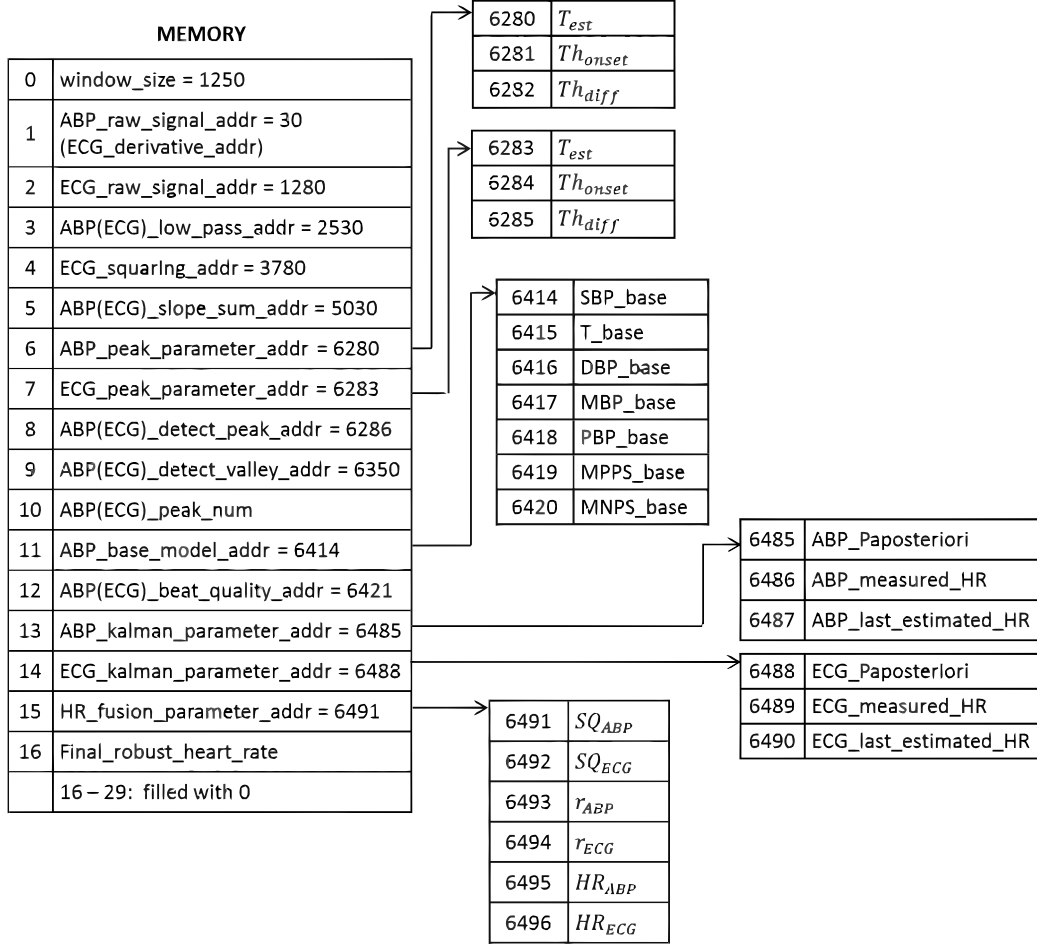


Figure 4.4: Memory layout of robust heart rate application in the hardware

and  $SQ_{ECG}$ ). In the end, the ABP and ECG features are passed to the FU10 (heart rate fusion) to compute the weighted heart rate. The computation processes are introduced in Chapter 3.

Figure 4.4 illustrates the memory layout of the application in the RRHMS hardware system. The numbers in the figure are the memory addresses. The memory system of the RRHMS is word-addressable, so each memory location is 4 bytes. The variables in the format of  $X\_addr$  are pointers to the memory address of  $X$ . So the inputs (raw ABP and ECG signals) are loaded into the memory locations of 30-1279 and 1280-2529, respectively, which are pointed to by the pointer variables in the memory locations of 1 and 2, respectively. The output (weighted heart rate) is stored in memory location 16. Some memory entries are shared for both ABP and ECG computations. For example, the variable in memory location 3 is a pointer to the low-



pass filtered values of both ABP and ECG signals. Therefore, both low-pass filtered values (with 1250 data points) of ABP and ECG signals are in the memory location of 1280-2529. Similarly, the other memory variables in parentheses in the figure are interpreted in the same way. The shared memory entries are usually used to store intermediate results in the ABP and ECG computations. Since the two signals are not processed in parallel, memories of the intermediate results are shared to reduce the memory footprint. As a result, the total amount of memory needed in the RRHMS is  $6497 \times 4 = 25,988$  bytes. So the 32 KB Block RAM in FPGA implementation or 32 KB SRAM in ASIC implementation is enough for the RRHMS memory system.

## Chapter 5

# RRHMS FAULT TOLERANCE DESIGN

This chapter introduces the low-overhead hardware fault detection and recovery mechanism used in RRHMS. We describe the hardware fault models and fault injections and then the hardware coverage of the proposed fault tolerance mechanism. Following that, the fault detection and recovery techniques are explained. At the end, the proposed fault tolerance technique is described, and various tradeoffs are discussed.

### 5.1 Fault Model and Injection

The fault model applied in this thesis is the independent low-level transistor fault. This fault may flip the result of its logic gate and then propagate to affect the application's output. We focus on transient fault detection and recovery. As for permanent faults, the proposed fault tolerance mechanism can detect them but cannot recover from them. To simulate faulty behavior in the hardware, we use fault injection at different levels, as follows:

- (1) *Application*: inject faults into the application code, such as flip a bit in the application variable or change the application control flows.
- (2) *Assembly/machine code*: change the bit in the instruction code, which may affect the instruction opcode, register value, branch direction, etc.
- (3) *Behavior hardware*: inject faults into the behavior hardware description code in Verilog/VHDL, such as flip the bit in the pipeline data path, in the program counter, in the decoder, etc.
- (4) *Hardware gate*: inject faults into the gate-level hardware code that is obtained from the hardware synthesis, such as flip the output of a logic gate.
- (5) *Transistor*: inject faults into the hardware transistors after the hardware synthesis and place-and-route, such as change the output current

or resistance of a transistor.

The *application* and *assembly/machine code level* fault injections are easy to implement and allow fast simulation, but they are only able to simulate faults that have a direct impact on the application. There are many low-level faults they cannot simulate, such as the fault that occurs in the hardware control logics and causes the hardware to hang in an infinite state loop. *Behavior hardware level* fault injection can simulate most low-level faults, but it is difficult to implement this fault injector to cover all the signal bits. This is because different signals written in the behavior hardware code have different formats, naming conventions, and bit widths. Therefore, the fault injection process (adding MUXs to the signal for fault simulation) cannot be automated. *Transistor level* fault injection best simulates the fault behavior that occurs in real life in the transistors. It can also realistically simulate how low-level transistor faults may propagate to affect the application results. However, the large amount of simulation time needed for analog level simulations (e.g., it may take days or even months to simulate just one second of hardware execution) makes it unfeasible for our experiment. As a result, we selected *hardware gate level* fault injection and inject random faults into the synthesized logic gates of the RRHMS. This choice is based on the following reasons:

- (1) It simulates low-level faults with high fidelity. Normally, a faulty transistor causes its logic gate (and, or, xor, etc.) to output the incorrect result.
- (2) The process is easy to automate, since the output of the synthesized logic gate is a single bit.
- (3) Even though the simulation time is long, it can complete in a reasonable amount of time for our purposes, i.e., in a matter of hours.

Since we focus on transient fault detection and recovery, we inject transient faults into the RRHMS hardware by randomly flipping the output of the synthesized logic gates. The logic gates are flipped independently at the adjustable fault injection rate. More details of the fault injection implementation can be found in Section 6.4.2.

## 5.2 Hardware Coverage

The low-overhead fault detection and recovery mechanism proposed in this thesis protects all the FUs in the RRHMS hardware system, as highlighted in the yellow box of Figure 5.1. Table 5.1 lists the percentages of the area, power consumption, and runtime of all the FUs in the hardware processing part for both FPGA and ASIC implementations. The hardware processing part (FU ASIC accelerator + MIPS controller) is the RRHMS hardware system without the shared on-chip memory. From the table, we see that the FUs account for more than 94% of the hardware area, more than 92% of the power consumption, and more than 97% of the application execution time. Therefore, the proposed low-overhead fault tolerance mechanism is able to protect most (or almost all) of the processing hardware for the robust heart rate detection application. The fault tolerance for the remaining part of the hardware system (MIPS controller, system coordination and configuration bus, memory arbiter, and shared on-chip memory) is not the focus of this thesis. Since the remaining processing hardware only accounts for small area

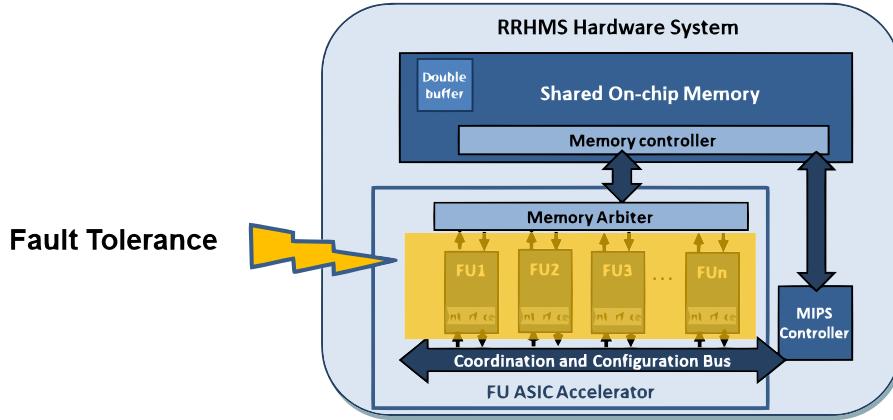


Figure 5.1: Hardware coverage of the proposed fault tolerance mechanism

Table 5.1: Percentages of all FUs in FPGA and ASIC implementations

	<b>FPGA</b>	<b>ASIC</b>
<b>Area (%)</b>	94.46	93.28
<b>Power (%)</b>	92.82	92.83
<b>Runtime(%)</b>	97.22	

Note: the percentages are for the processing hardware part (RRHMS hardware except memory: FU ASIC accelerator and MIPS controller).

and power percentages, simply applying double or triple modular redundancy (DMR or TMR) to enhance their reliability does not introduce much area and power overhead (less than 10% in DMR and less than 20% in TMR). In addition, the on-chip memory can be protected by applying the popular error-correction coding (ECC) technique [102].

### 5.3 Fault Detection

FU hardware faults are detected using *heartbeats* and *invariant checking* [26] techniques in the hardware. The heartbeat is recorded by the heartbeat counter to detect FU hangs during the execution. In the RRHMS, normal FU executions never result in FU hangs. Therefore, an FU hang is an indicator of a hardware fault. One possible cause of FU hangs is the hardware fault that occurs in the control logics of the FU state machine controller and results in an infinite state loop. When the FU execution starts, the heartbeat counter is reset and starts to count the number of cycles of the FU execution. There is a timeout register for each FU that can be configured to set the limit of the cycle number allowed for the FU’s execution. In each cycle during the execution, the value of the heartbeat counter is checked with the corresponding FU timeout register. If the value of the heartbeat counter is larger than the value set in the timeout register, an FU hang, and hence an FU fault, is detected.

In addition to heartbeat, the invariants are checked to detected faults that may not result in FU hangs but do cause the FU to behave incorrectly in writing results. According to [26], program invariants are the conditions that hold true during the program’s execution. Therefore, if any invariant is violated, it means a fault has occurred. Two kinds of invariants are used in the proposed fault detection mechanism: *address invariants*, which are in the FU result writing address, and *result invariants*, which are in the FU result.

**Address invariants** are obtained during the compiling of the application, when the compiler assigns the memory locations at which each FU will write its final computation results. Since in our prototype we write the assembly code for the application, the memory locations of FU results are manually set, and the address invariants are obtained directly during the assembly implementation of the application. Some FUs only generate a single

Table 5.2: FU result invariants, profiled with 1000 windows (2.78 hours) of patient data (a40050) from the MIMIC II database

Functional Unit		Result Invariant			
		<i>min</i>	<i>max</i>	<i>min<sub>diff</sub></i>	<i>max<sub>diff</sub></i>
FU0 - low-pass	for ABP	32.40	176.10	-18.60	23.40
	for ECG	-6.32	7.25	-4.16	-4.16
FU1 - slope sum	for ABP	0.00	87.30	-20.70	52.20
	for ECG	0.00	18.18	-16.62	18.18
FU2 - peak detection	peak index	0.00	1240.00	45.00	322.00
	peak number	1.00	17.00	-8.00	13.00
FU3 - heart rate		53.35	104.53	-37.19	40.40
FU4 - Kalman filter	filtered value	26.35	100.55	-25.09	17.99
	residue	-92.87	66.60	-	-
FU5 - signal quality		0.00	1.00	-	-
FU6 - derivative		-4.16	4.02	-6.38	8.18
FU7 - squaring		0.00	17.31	-11.14	16.62
FU8 - ABP beat quality		0.00	1.00	-	-
FU9 - ECG beat quality		2.16	8.98	-	-
FU10 - heart rate fusion		86.73	100.51	-1.50	2.58

Note: the following conditions hold true during the corresponding FU execution:  $y_i \geq \min$ ,  $y_i \leq \max$ ,  $y_i - y_{i-1} \geq \min_{diff}$ , and  $y_i - y_{i-1} \leq \max_{diff}$ , where  $y_i$  is the current result value and  $y_{i-1}$  is the previous result value.

result, such as FU3 (heart rate), FU5 (signal window quality), FU10 (heart rate fusion), and others. So the result address for each of them is an exact memory location, and this is used as the corresponding FU's address invariant. The other FUs generate result arrays instead of single results, such as FU0 (low-pass filter), FU1 (slope sum), FU2 (peak detection), and others. The result address for each of them is a memory range. Additionally, since they write the result elements one by one from the lower index, the memory addresses of two consecutive writes differ only by one. Therefore, for these FUs, both the result memory ranges and address differences between consecutive writes serve as their address invariants. Therefore, if an FU tries to write its result to a memory location other than the one it is assigned, the cause must be a hardware fault in the FU. Such behavior is considered an address invariant violation and a hardware fault indicator. Once the application is compiled, the FU address invariants do not change with changing patient data. However, this is not the case for the result invariants.

**Result invariants** are obtained by application profiling using patient data. Since each FU is responsible for a processing step in the heart rate

detection flow (Figure 3.1), the output of each FU has a specific high-level application meaning. This contrasts with the output of basic instructions, like addition, and subtraction, in a general-purpose processor. Therefore, meaningful patterns can be found in the FU results of the proposed hardware system. For example, the results of the low-pass filter FU are the biomedical signals with high-frequency noise removed. So they must be within a meaningful range, based on physiological limitations, such as the highest and lowest possible blood pressures. In addition, since the biomedical signals are sampled at a specific frequency (125 Hz for our data), the difference between two consecutive low-pass filtered data should be within another range, according to the slope of the signal waveform. We use this property to obtain the FU result invariants to detect FU faults. The FU result invariants are profiled with the target patient’s data for patient-specific fault detection. Table 5.2 lists the example result invariants of the 11 FUs in the RRHMS, obtained by profiling 1000 windows (2.78 hours) of patient data (a40050) from the MIMIC II database. Similar to address invariants, both the result range and difference between consecutive computed results are profiled and used as result invariants. The first two columns of the result invariants in Table 5.2 are the minimum and maximum values of the corresponding FU results (lower and upper bound of the result range), and the other two columns of result invariants are the minimum and maximum values of the difference between consecutive computed results. Both of the two kinds of result invariants are not profiled for every FU, as some of them are meaningless for fault detection. For example, the Kalman residue value computed in FU4 and signal quality value computed by FU5 represent the quality of the signal in the current window. So the difference between the consecutive results of those FUs indicates the difference of the signal qualities between consecutive windows, which depends on the bio-sensors. Using them as result invariants may mistakenly detect sensor changes as FU faults. This also explains the missing result invariants of FU8 and FU9, which evaluate signal beat qualities. The result invariants are signal- and patient-specific, similar to some FU parameters (like threshold parameters for peak detection). Therefore, they are designed to allow configurations using the same bus interface as the FU configuration registers.

With the heartbeats and invariant checking, the proposed fault detection mechanism is able to detect the hardware faults (both transient and perma-

ment) that cause the FU to:

- (1) hang (enter infinite state loop and unable to finish execution within the amount of time specified by its timeout register),
- (2) write result to a memory location that violates its address invariants, or
- (3) produce incorrect results that violate its result invariants.

The FUs in the existing hardware system do not need to be modified for the proposed fault detection mechanism or for the proposed recovery mechanism introduced in the next section. The proposed fault detection and recovery are handled by a specialized hardware module, the fault detection and recovery unit (FDRU), depicted in Figure 5.2. The FDRU sits in the FU ASIC accelerator. It both monitors the system coordination and configuration bus for FU commands (configuration and execution) and listens to the memory arbiter for FU memory requests.

Figure 5.3 illustrates the operation flow of FDRU. When no FU is executing, FDRU is in *State 1*, waiting for FU instructions sent from the MIPS controller. When the FU configuration instruction (FU\_CFG) is sent, FDRU stores these values for the FU configuration register at the time of the configuration of the corresponding FU (*State 2*). This way, FDRU keeps a synchronized copy of the configuration register values of all FUs; these values are used for the recovery process. Upon receiving the FU execution instruction (FU\_START) from the MIPS controller, FDRU enters *State 3* and starts

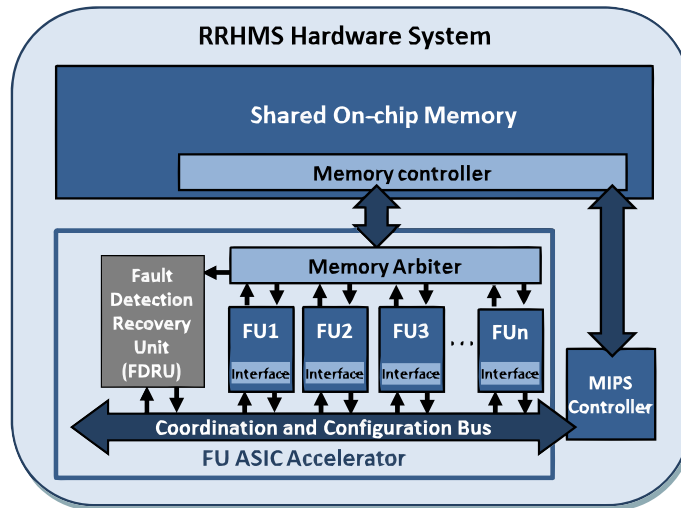


Figure 5.2: Fault detection and recovery unit (FDRU) in hardware system



the heartbeat counter. During the FU execution, FDRU stays in *State 3*, where it checks the heartbeat counter for FU hang detection and monitors the FU memory write request address and result invariants. When there is either an invariant violation during the FU memory write or a time out in the heartbeat counter, FDRU starts the fault recovery process. If neither is detected and the FU finishes normally (sends a *DONE* signal to the system coordination and configuration bus), FDRU goes back to *State 1* to wait for new FU instructions.

## 5.4 Fault Recovery

To recover from a detected FU fault, the corresponding FU is re-executed after being reset and reconfigured. If the fault is transient, re-execution is enough to recover from it, since the same fault is unlikely to occur in the re-execution. However, if the detected fault is permanent, re-execution will result in detection of the same fault. After three re-executions, if the fault

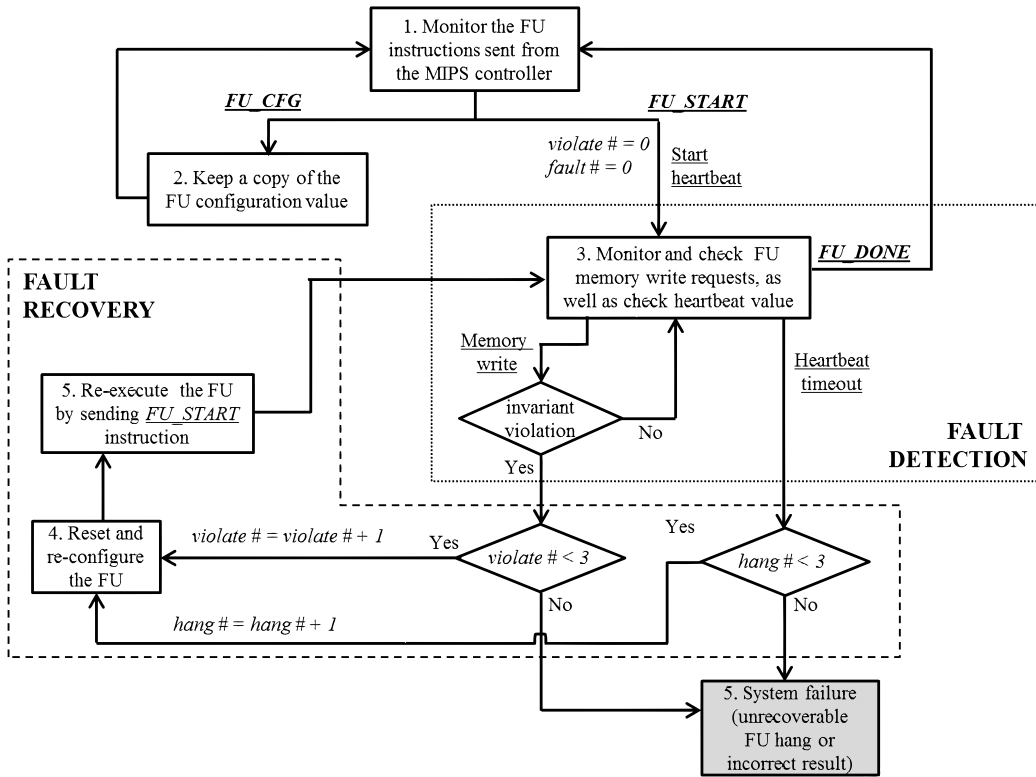


Figure 5.3: Fault detection and recovery unit (FDRU) operation flow

still exists, the application is aborted and the user is notified. The proposed fault recovery mechanism is able to recover from the detected transient fault, but not from the permanent one.

Similar to the fault detection mechanism, the proposed fault recovery mechanism does not need FU modifications; it is also handled by FDRU. On detecting an FU fault, FDRU starts the fault recovery process. For either invariant violation or heartbeat timeout, three chances are allowed. This means that if, in a single FU execution, an invariant violation is detected three times or an FU hang is detected three times, the fault is considered to be unrecoverable, and the application is aborted and the user notified. This phenomenon may be caused either by a permanent fault or by a transient fault that continually occurs in re-executions, signifying that the transient fault rate is high. As shown in Figure 5.3, FDRU maintains two variables, *violate#* and *hang#*, to track the number of times each kind of fault has occurred. Both variables are reset to 0 when the FU is about to start execution and before FDRU enters *State 3*. If the detected fault has not occurred three times in the current execution, the corresponding variable is incremented for this detection and FDRU goes to *State 4*. Here it resets the FU to stop the current execution by sending it the reset signal and reconfigures all configuration registers of the FU with the copy it has stored in *State 2*. The FU reconfiguration is achieved by the FU configuration instructions (FU\_CFG) sent from FDRU. This is necessary because the fault may corrupt some configuration register values, either directly in the configuration register bits or indirectly by propagating to change the configuration register values. After reset and reconfiguration, FDRU enters *State 5* to restart the FU execution by sending it the FU execution instruction (FU\_START), after which it goes back to *State 3* and monitors the re-execution in the same way.

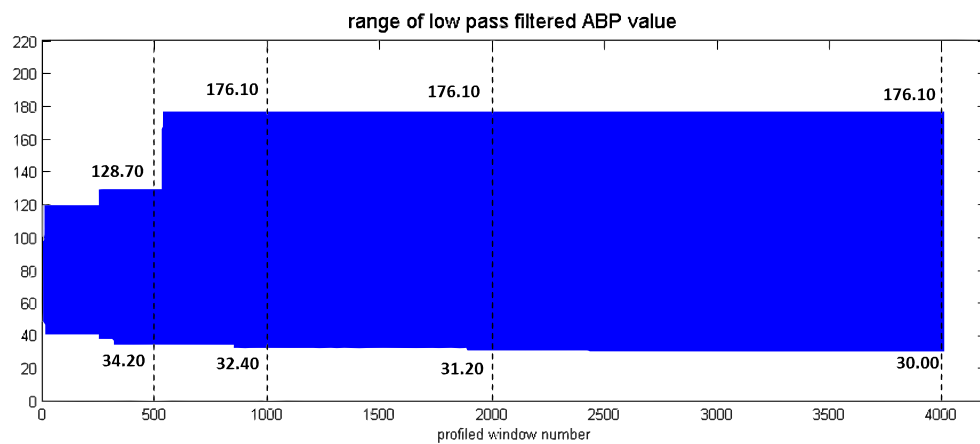
Therefore, FDRU is designed for the existing hardware architecture by using the same system bus used by the MIPS controller for FU operations to implement fault recovery. It also uses the memory arbiter design that collects the FU memory requests to do invariant checking. As a result, no major modification is needed in the baseline hardware system to incorporate FDRU to enable fault tolerance in FUs. In addition, the overhead of FDRU is low. For hardware area (resource) overhead, besides the simple state logics (Figure 5.3), FDRU only needs to maintain (1) a copy of the values of all FU configuration registers (Table 4.1), (2) timeout registers for each FU and

a heartbeat counter, (3) address and result invariants of each FU (Table 5.2) and invariant checkers (hardware comparators). As for performance, FDRU does not introduce any overhead. Its operations, shown in Figure 5.3, are completely in parallel with the baseline hardware executions. Any power overhead results from the parallel updates of configuration registers and fault detections (invariant and heartbeat checking). This overhead is much smaller than the power consumption of the baseline hardware. In addition, the FDRU module does not decrease the reliability of the baseline RRHMS hardware system. If FDRU fails, the baseline system continues to operate normally, just as if there were no fault tolerance protection.

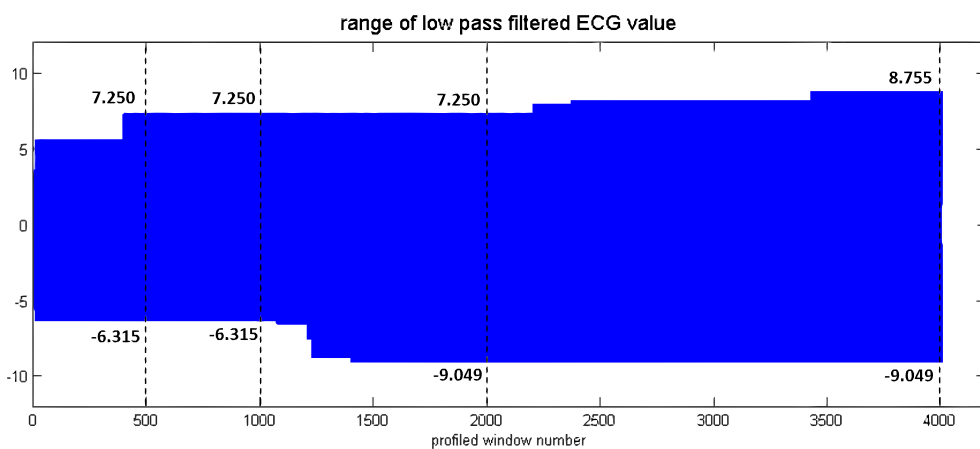
## 5.5 Fault Tolerance Coverage Discussion

The proposed fault detection and recovery mechanism with FDRU has much smaller area and power overheads than the traditional fault tolerance technique using double modular redundancy (DMR), which is only able to recover from transient fault through re-execution. The area and power overheads in DMR are each over 100% because the hardware modules and operations need to be duplicated. The detailed FDRU overhead results are discussed in Section 6.4.1. The tradeoff for this small overhead is in fault detection coverage. DMR is able to detect any fault that causes a result mismatch between duplicate executions. FDRU is only able to detect faults that result in an FU hang, address invariant violation, or result invariant violation. But a fault may lead to an incorrect FU result without causing an FU hang or violating any invariant conditions. For example, if the result invariants listed in Table 5.2 are used, a fault that occurs in FU0 during ABP computation may generate a result that (1) is written to the correct memory location, (2) is within the range of  $[32.40, 176.10]$ , and (3) has a difference within the range of  $[-18.60, 23.40]$  between its consecutive results (the results before and after it). Nonetheless, this incorrect low-pass filtered result may cause incorrect peaks to be detected and result in an incorrect final heart rate estimation. The proposed fault detection mechanism in FDRU cannot detect this kind of fault. Fortunately, this kind of fault is rare, even under a high fault rate, as shown in Section 6.4.3.

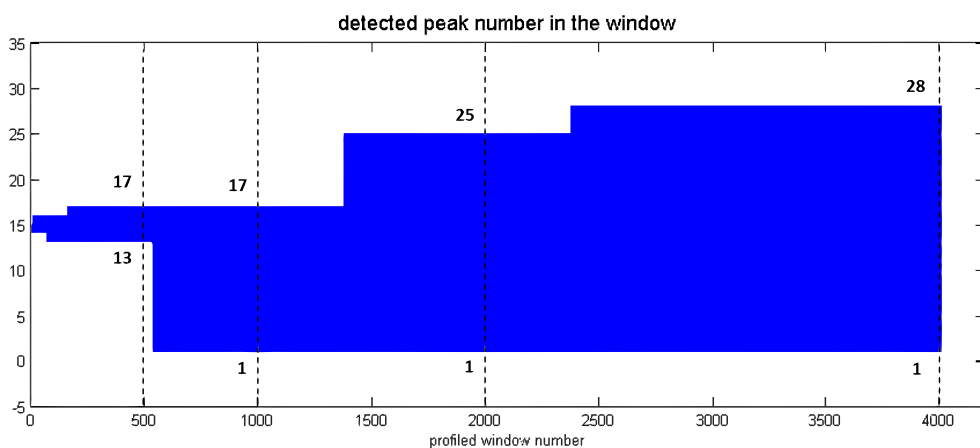
In addition, fault detection by FDRU depends on profiling of the result



(a) Range of the low-pass filtered ABP values



(b) Range of the low-pass filtered ECG values



(c) Range of the detected peak number in a window

Figure 5.4: Relationship between result invariants and profiled window number with patient (a40050) data from the MIMIC II database (the range becomes larger with more profiled data)

invariants. It should be noted that there is a tradeoff between fault detection coverage and the false positive rate. A false positive may be caused by a non-profiled scenario that causes a result invariant violation. For example, abnormal signal inputs may cause out-of-range low-pass filtered results, not just by hardware faults. If this particular abnormal scenario did not occur during invariant profiling, the invariant range may not include the corresponding results. The basic problem is that if result invariants are set too tightly (include only the few profiled data), FU faults caused by new data scenarios may generate false alarms; on the other hand, if they are set too loosely (include the full range of all possible integer values), invariant checking becomes useless.

This tradeoff is illustrated in Figure 5.4, which shows three examples of the relationship between the profiled invariant range and number of profiled windows. The blue area in the figure is the range of the invariant profiled with the number of windows specified by the x-axis value. The lower edge of the blue area is the lower bound of the invariant range, and the upper edge is the upper bound. As the number of profiled windows increases, the profiled invariant range becomes larger (the lower bound becomes smaller, and the upper bound becomes larger) because more data scenarios are included. For example, in Figure 5.4a, if only the first 500 windows of patient data are profiled, the invariant range obtained for the low-pass filtered ABP value (result of FU0) is  $[34.20, 128.70]$  ( $range_1$ ). If the first 1000 windows of patient data are profiled, the range is  $[32.40, 176.10]$  ( $range_2$ ). When 4000 windows (11.11 hours of patient data) are profiled, which include 11 occurrences of arrhythmia problems, the range becomes  $[30.00, 176.10]$  ( $range_3$ ). Therefore, if  $range_1$  is used as the result invariant of FU0 for ABP processing, the result of 176.10 caused by the new input scenario will be mistakenly detected as an FU fault. On the other hand, if  $range_3$  is used as the invariant, the detection coverage becomes smaller. In this case, the result of 176.10 when processing the first 500 windows must be caused by the FU fault, but since it does not violate any invariant range, it will go undetected. Section 6.4.4 analyzes detection coverage with different invariant ranges obtained by profiling different numbers of windows. Since the probability of the above undetected case is low, profiling should include as much data as possible to prevent false positives.

However, sometimes it is impossible to include all scenarios in profiling be-

cause the signal changes and patient problems are rare in real life. Therefore, to reduce false positives, the doctor's knowledge of the patient can be combined with profiling to set the result invariants in some FUs whose invariants depend on the input signals, such as low-pass filter (FU0), slope sum (FU1), heart rate (FU3), and others. For those FUs, the patient's physiological limitations can be used to find the invariants (e.g., the patient's heart rate never exceeds 250 bpm nor drops below 30 bpm). The other FUs do not have this problem, such as signal quality (FU5), ABP and ECG beat qualities (FU8 and FU9), and others. For example, the result of FU5 (percentage of good beats in the window) should always be between 0 and 1, no matter what the inputs are.

A false positive, on the other hand, is not always harmful. Sometimes it is an indication of severe signal corruptions that may be caused by sensor disconnection. If the invariants are set by profiling long periods of data with the patient's physiological limitations considered, and if a detected fault is not actually caused by a hardware fault, then it is certain that the detected fault is caused by input that is beyond the patient's physiological limitation, which is very likely due to severe signal corruption and is worth the user's attention.

# Chapter 6

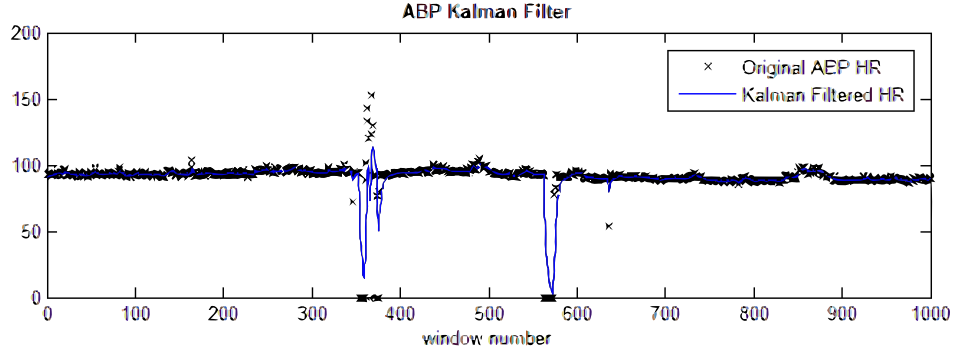
## EXPERIMENTAL RESULTS

This chapter starts with a discussion of the accuracy of the heart rate detection algorithm introduced in Chapter 3, followed by an evaluation the proposed RRHMS hardware system implemented on both ASIC and FPGA platforms. The same application is also implemented on a Nexus 7 tablet for comparison with the embedded processors on the market. At the beginning of the hardware evaluation, experiment setups are introduced for the implementations on the three platforms (Android, FPGA, and ASIC). Then the baseline RRHMS without fault tolerance features is evaluated, and the three platform implementations are compared. Then follows an evaluation of the proposed fault tolerance technique using FDRU, where the overheads and fault tolerance coverage under injected hardware faults are discussed.

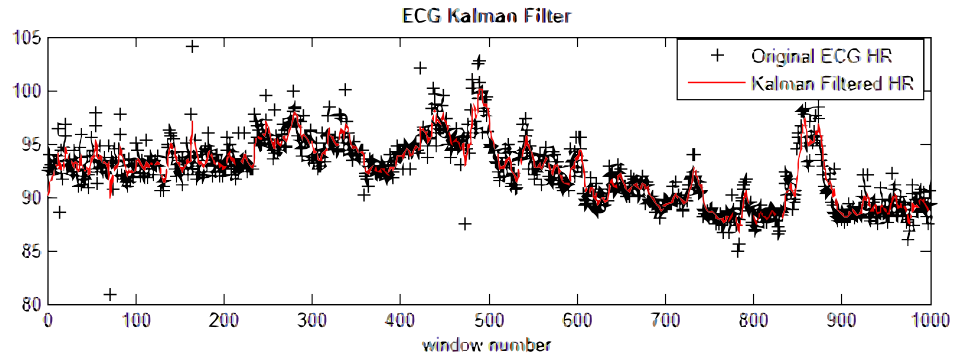
### 6.1 Heart Rate Detection Accuracy

The heart rate detection algorithm introduced in Chapter 3 is implemented in MATLAB to evaluate its accuracy. The hardware implementations are compared with the results of the MATLAB code to check the implementation correctness. We first compared the algorithmically optimized ABP and ECG beat detection algorithm with the original algorithms [18] and [28] for ECG and ABP beat detections, respectively. We used the patient data from the MIMIC II database with 24 patients, totaling 270.9 hours of ECG and ABP data. Both ECG and ABP beats detected by our optimized algorithm match the results of the original algorithms. In addition, we manually inspected several segments of the raw data with the detected beats marked, and all the inspected detected beats match our manual annotations.

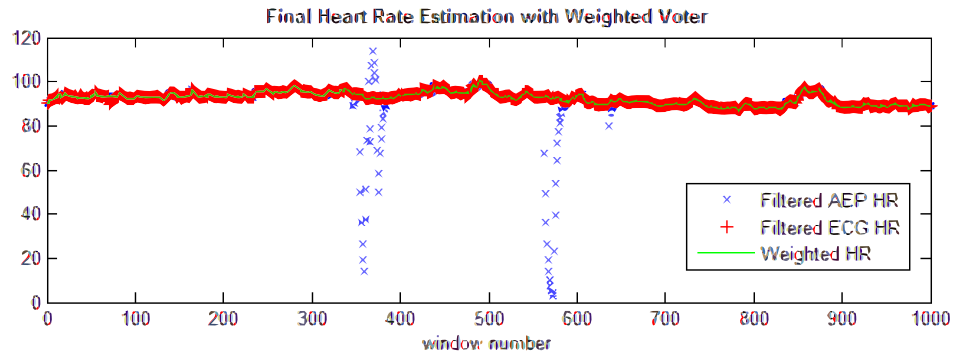
The heart rate fusion algorithm that we applied, by weighting on the signal qualities and Kalman residues, was proposed and validated in [21]. Our ex-



(a) Kalman filtered ABP heart rates



(b) Kalman filtered ECG rates



(c) Weighted heart rate

Figure 6.1: Kalman filtered ABP and ECG heart rates for 1000 windows, and final weighted heart rates of the corresponding windows (data from the MIMIC II database patient a40050)



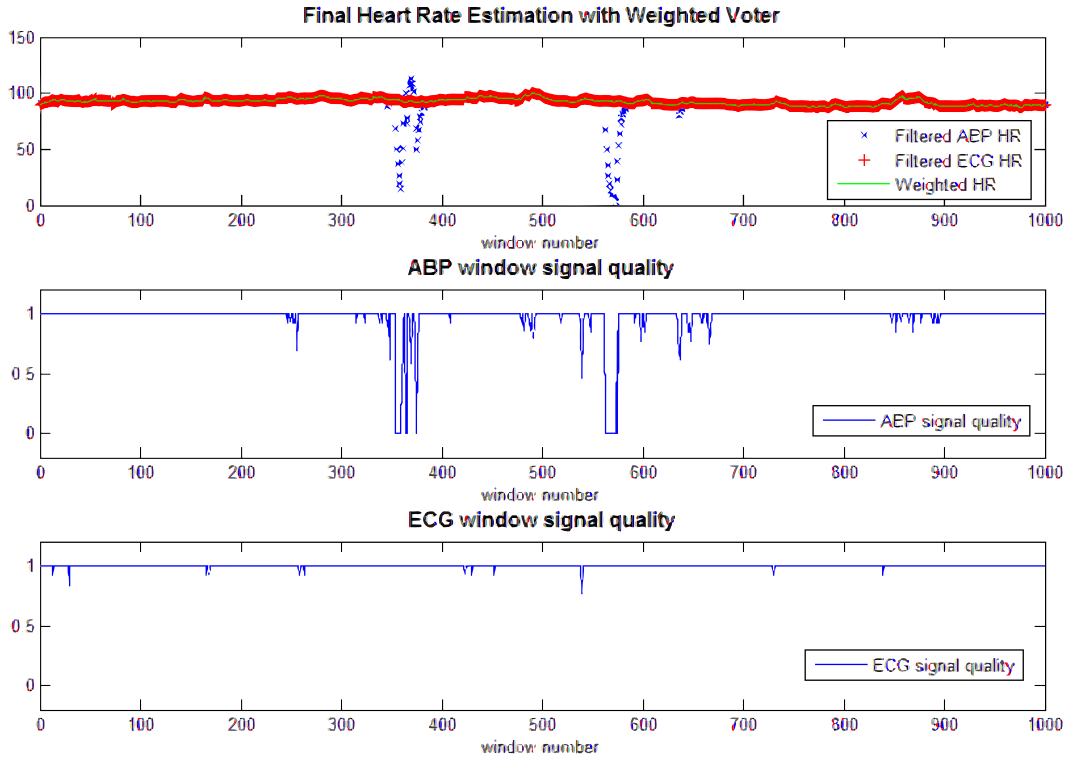


Figure 6.2: ABP and ECG signal qualities (the corresponding signal qualities that are used to obtain the heart rate fusion results in Figure 6.1)

periments also prove its effectiveness. The heart rate fusion examples shown in Figure 3.4 and Figure 3.5 illustrate that the fusion algorithm is able to provide continuous robust heart rate monitoring even when one signal is corrupted by noise and artifacts. In addition, Figure 6.1 provides another example with 1000 windows of patient (a40050) data, where the ABP signal is corrupted around window 360 and window 570, and the fusion algorithm fixes the ABP corruption by weighting more on the better quality ECG signals during the two window segments. Figure 6.2 explains the reason behind the effectiveness of the fusion algorithm in Figure 6.1 by showing and comparing the signal qualities of ABP and ECG for the corresponding windows. The figure shows that at around window 360 and window 570, ABP quality becomes low while ECG quality is high. Therefore, more weight is put on the heart rate extracted from the ECG signal.

## 6.2 Hardware Experiment Setup

The proposed RRHMS is implemented both in ASIC design using the Synopsys Design Compiler and on the Xilinx FPGA platform. ASIC is the target platform for the final product of the RRHMS to provide heart rate monitoring with low energy. FPGA is the platform that uses the proposed hardware system as a framework for efficient embedded system designs (by adding and removing FUs for the target embedded application). Therefore, both ASIC and FPGA implementations are evaluated in this thesis. For comparison with the embedded processors on the market, the same heart rate detection application is implemented as an Android application on the Asus Nexus 7 tablet (2013 model).

Table 6.1 lists the experiment setups of the three platforms: Android, FPGA, and ASIC. For the Android implementation, the Android software development kit downloaded from the Android developer’s website [103] is used to compile and load the application to the target hardware, a Nexus 7 tablet. The Nexus 7 tablet is equipped with the Qualcomm Snapdragon S4 chipset running at 1.5 GHz. The Android application on the tablet runs in the Krait processor on the chipset, which is architecturally similar to the ARM processor. The execution time of the application is recorded by inserting time measure functions in the code before and after the heart rate detection

Table 6.1: Experiment setup on the three platforms

Platform	Frequency	Design Tools	Evaluation Tools
Android	Snapdragon S4 @ 1.5 GHz	Android SDK [103] (test and evaluate on the 2013 Asus Nexus 7 tablet)	Qualcomm Trepro Profiler [104] and
FPGA	66.6 MHz	Xilinx ISE Design Suite 14.2 (test and evaluate on the Virtex 5 ML507 (XC5VFX70T) FPGA board)	Modelsim SE 10.1a, Xilinx ISE, and Xilinx Power Analyzer
ASIC	100 MHz (up to 222.2 MHz)	<b>Processing logics:</b> Synopsys Design Compiler [27] with NanGate 45 nm Open Cell Library [105]. <b>On-chip memory:</b> Synopsys Generic Memory Compiler (32 nm) [106]	Modelsim SE 10.1a and Synopsys Design Compiler with NanGate 45 nm and Generic Memory Compiler 32 nm SRAM libraries [105, 106]

portion. The power consumption on the Android platform is profiled using the Qualcomm Trepro Profiler [104]. To obtain accurate power consumption of only the heart rate detection, the baseline power of the application without starting heart rate detection is measured first. Then the total power is measured with the heart rate detection executing. The power difference between the two measurements is the power consumption of the heart rate detection on Android. During both execution time and power measurements, all the other Android applications and services are turned off.

The Xilinx Virtex 5 ML507 board (XC5VFX70T) is used for FPGA implementation. The RRHMS run on the actual FPGA board, while the reported results are collected from the simulation of the FPGA-synthesized design of the hardware. The Xilinx ISE Design Suite is an integrated Xilinx software platform for Xilinx FPGA synthesizing, mapping, and routing. We use the 14.2 version of Xilinx ISE to implement the RRHMS hardware that is written in VHDL and Verilog. The processing logic of the hardware system (non-memory part: FU ASIC accelerator and MIPS controller) is implemented in the FPGA logics, and the on-chip memory is implemented as Block RAM on the FPGA board. The application's execution time on FPGA is calculated by the multiplication of the hardware cycle number and clock period. The hardware cycle number is obtained from cycle-accurate simulation in Modelsim SE 10.1a, and the clock period is obtained from the timing report generated by Xilinx ISE after FPGA routing. In the FPGA implementation, the maximum clock frequency achieved is 66.6 MHz (15 ns clock period). The power consumption of FPGA is profiled using the Xilinx Power Analyzer. To obtain accurate power results, signal activities are collected from the post-routing simulation in the Modelsim simulator. With the signal activities provided, the Power Analyzer shows high confidence in the profiled power number, which we use as the FPGA power consumption.

In ASIC implementation, separate tools are applied to synthesize the on-chip memory and processing logic (FU ASIC accelerator and MIPS controller) of the RRHMS. The Synopsys Design Compiler [27] is used to synthesize the processing logic with the 45 nm NanGate Open Cell Library [105]. The Synopsys Generic Memory Compiler [106] is used to synthesize the on-chip memory using 32 nm SRAM technology. (There is no 45 nm SRAM in the Generic Memory Compiler, and 32 nm is the closest available technology process to the 45 nm used for the processing logic.) The ASIC implementa-

tion of the RRHMS is only simulated (not tested on the real taped-out chip). Similar to FPGA implementation, the execution time of ASIC is obtained by the multiplication of the hardware cycle number from Modelsim simulation and the clock period from the ASIC synthesis process. To compare the three platforms, the 100 MHz clock frequency of ASIC is chosen; its maximum synthesized clock frequency is 222.2 MHz. The power of ASIC is profiled from the tools in the Design Compiler, which takes three kinds of input files for accurate power estimation: (1) synthesized netlist files, (2) technology library files (NanGate 45 nm cell library and Generic Memory Compiler 32 nm SRAM library), and (3) signal activities (processing logic and memory activities) collected from the post-synthesis simulation in Modelsim.

## 6.3 Baseline RRHMS Evaluation

This section evaluates the baseline RRHMS without fault tolerance features. The runtime performance, power, and energy consumptions of the Android, FPGA, and ASIC implementations are compared. Then, the total resource utilizations of the FPGA and ASIC implementations are introduced. Finally, the breakdowns of runtime, resource utilization, and power consumption of the processor logics (FU ASIC accelerator and MIPS controller) on the FPGA and ASIC platforms are discussed.

### 6.3.1 Comparison of Android, FPGA, and ASIC

The application results of Android, FPGA, and ASIC implementations are compared with MATLAB simulations to ensure the implementation correctness. For comparison, 1000 windows of patient data (both ABP and ECG data collected at 125 Hz frequency) are processed for robust heart rate estimation. Table 6.2 lists the results of runtime performance, power, and energy consumption of the ASIC implementation to process the 1000 windows of data. Figure 6.3 compares the different results on the three platforms (all platform results are normalized to the results of ASIC).

As shown in Table 6.2, ASIC finishes processing 1000 windows of data in only 0.211 s. So on average, ASIC only takes about 0.211 ms to process a single window containing 10 s of patient data. Therefore, for each 10 s of

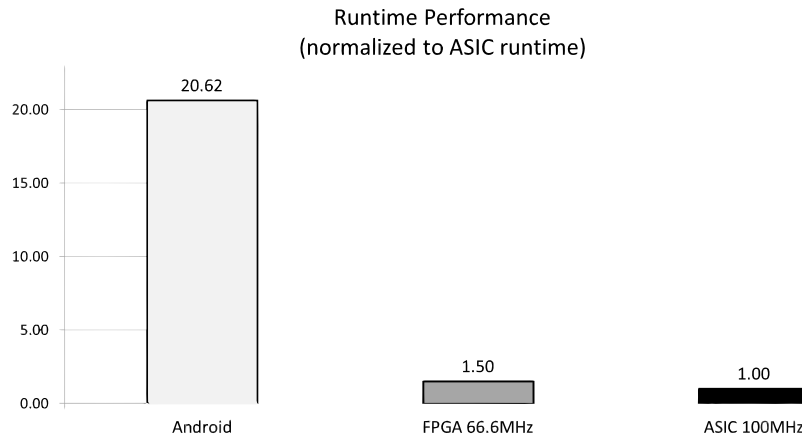
Table 6.2: Runtime, power, and energy results of ASIC implementation (run of 1000 windows of patient data (a40050) from the MIMIC II database)

	Runtime (s)	Power (mW)	Energy (mJ)
<b>Processing logic*</b>	-	6.887	1.453
<b>On-chip memory</b>	-	0.4541	0.0958
<b>Total</b>	0.211	7.341	1.549

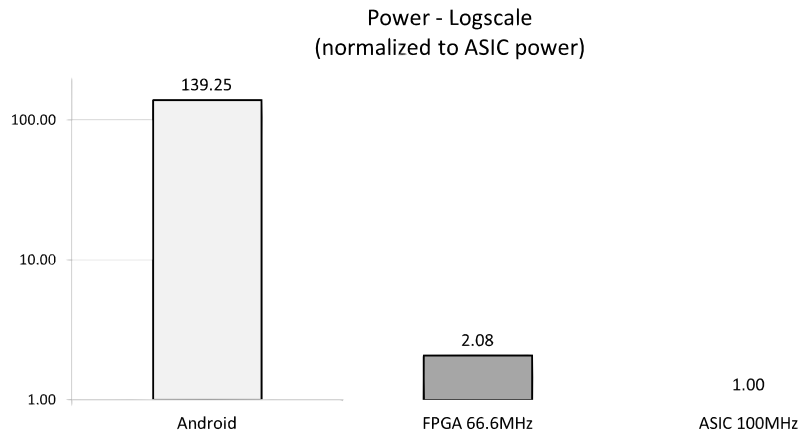
\*Processing logic: FU ASIC accelerator and MIPS controller (non-memory part).

patient data collected from the sensors, the ASIC implementation can estimate an average heart rate in 0.211 ms (active in 0.00211% of the time). As a result, ASIC works in the duty cycle manner to save energy consumption. It only needs to be woken up every 10 s when a new window of data is ready. After processing, it can be put in sleep mode with the power gating technique. The power consumptions of the processing logic and on-chip memory are profiled separately. The power number reported in the table is the active power consumption, including both dynamic and static (leakage) powers, during the processing of the 1000 windows of data. Most of the power (93.82%) is consumed in the processing logic because the computation is not memory-intensive and the memory is synthesized with a more advanced technology library (32 nm). In total, the ASIC active power consumption during the processing is only 7.341 mW. “Energy consumption” in the table means the total energy consumed to process the 1000 windows of data. It is calculated by multiplying the active power consumption with the runtime. The total energy consumed to process 1000 windows (10,000 s) of patient data is only 1.549 mJ. Therefore, if the ASIC implementation is duty cycled with power gating, the approximate average power consumption during daily monitoring is calculated as  $1.549 \text{ mJ} / 10,000 \text{ s} = 0.1549 \text{ } \mu\text{W}$  (patient data is sampled at 125 Hz).

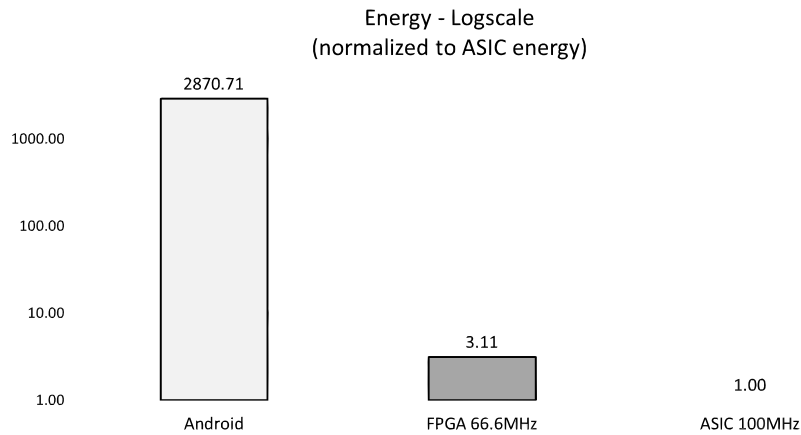
Compared with ASIC, the execution times needed to process the same amount of data (1000 windows) on Android and FPGA platforms are 20.62 and 1.50 times longer, respectively, as illustrated by Figure 6.3a. The speedup of ASIC compared to Android implementation is mainly due to: (1) the efficiency of the RRHMS FU modules that are optimized with ASIC logics, and (2) the faster memory access with the on-chip memory design. On the other hand, since the same underlying register transfer level (RTL) hardware



(a) Runtime performance comparison



(b) Power consumption comparison



(c) Energy consumption comparison

Figure 6.3: Comparison of the Android, FPGA, and ASIC implementations (run of 1000 windows of patient data from the MIMIC II database)

design is used in both the FPGA and ASIC implementations, the hardware cycles to run the same application with the same input data on these two platforms are also the same. The speedup of ASIC compared to FPGA is due only to the higher clock frequency.

ASIC also achieves the lowest power consumption in the comparison (Figure 6.3b). The power consumption compared in the figure is also the active power during the processing of the data. The power consumption of Android is 139.25 times that of ASIC, while the power consumption of FPGA is 2.08 times that of ASIC. This is because ASIC has the least hardware complexity. The Android implementation runs on the Krait processor of the Nexus 7 tablet, a general purpose embedded processor that has complicated processing pipelines and hierarchical memory systems. It is designed to reduce the processing latency of general applications. On the other hand, the ASIC implementation of the RRHMS hardware system is composed of FUs designed especially for computation of the target application, and it is directly synthesized with logic gates. Furthermore, due to the small memory size required, the on-chip memory implemented with SRAM is directly used as the main memory of ASIC. This greatly simplifies the memory system and reduces the corresponding power consumption. The FPGA implementation benefits from the same FU designs to efficiently support computations and from the small memory footprint to reduce the complexity of memory system. As a result, FPGA also achieves good power efficiency (about 1/67 of Android's power). However, FPGA logics are realized with look-up tables instead of logic gates, which adds some complexity and overhead. Therefore, the FPGA implementation consumes more power than the ASIC implementation does.

The energy comparison in Figure 6.3c compares the total energy consumed to process the 1000 windows of data on the three platforms. Energy consumption, calculated by multiplying execution time and active power consumption, directly affects the battery life. Since compared with Android, both FPGA and ASIC implementations can finish the processing in less time and at smaller power consumptions, they both achieve much better energy efficiency than Android (about 1/923 and 1/2871 of Android's energy consumption, respectively). Therefore, given the same battery capacity, the FPGA and ASIC platforms of the RRHMS are able to achieve as much as 923 and 2871 times more battery life of the Android platform, assuming the inactive power is negligible, which can be achieved with power gating in the sleep mode.

In summary, the comparison results show that the proposed RRHMS in both FPGA and ASIC implementations is much more energy-efficient than the general-purpose embedded processor, such as Krait, to provide real-time heart rate monitoring. This enables RRHMS to achieve longer battery life, as well as portability, by not requiring a big, heavy battery to support it.

### 6.3.2 RRHMS Resource Utilization

Table 6.3 lists the resource utilizations of RRHMS in the FPGA and ASIC implementations. In the FPGA implementation, resource utilization data are obtained from the module-level utilization report generated from the FPGA mapping process in Xilinx ISE. The total processing logic occupies 11,856 look-up tables (LUTs) and 22 DSP48E slices of the FPGA resource. The RRHMS on-chip memory is implemented using the 32 KB Block RAM on the FPGA board. In the ASIC implementation, the processing logic is synthesized with 53,697 cell gates and occupies 0.121 mm<sup>2</sup> of the die area. The on-chip memory in ASIC is synthesized as 32 KB SRAM with the 32 nm library of the Generic Memory Compiler (different from the 45 nm cell library for processing logic), which accounts for 0.195 mm<sup>2</sup> of the die area.

Table 6.3: Resource utilizations of the RRHMS on FPGA and ASIC

	<b>FPGA</b>	<b>ASIC</b>
<b>Processing logic*</b>	11,856 LUTs** + 22 DSP48E	53,697 cell gates (0.121 mm <sup>2</sup> )
<b>On-chip memory</b>	32 KB Block RAM	32 KB SRAM (0.195 mm <sup>2</sup> )

\*Processing logic: FU ASIC accelerator and MIPS controller (non-memory part).

\*\*LUT means look-up table (LUT is 6-input for the Virtex 5 FPGA family).

### 6.3.3 Runtime, Resource, and Power Breakdown

Table 6.4 lists the breakdown of the runtime, resource utilization, and power consumption of the processing logics (FU ASIC accelerator and MIPS controller) in both FPGA and ASIC implementations. The breakdown of each result is in terms of percentage. The absolute result of each part in the table can be calculated with the total execution time, resource utilization, and power consumption of the corresponding platform, which can be found



Table 6.4: Runtime, resource utilization, and power breakdown

		Runtime (%)	Resource (%)		Power (%)	
			FPGA	ASIC	FPGA	ASIC
FUs	0: low-pass filter	<b>11.88</b>	4.08	3.76	10.65	4.68
	1: slope sum	11.84	7.64	5.74	<b>23.79</b>	9.08
	2: peak detection	<b>24.32</b>	<b>14.65</b>	7.14	8.67	5.78
	3: heart rate	1.00	4.64	3.24	0.25	4.13
	4: Kalman filter	0.29	6.34	9.77	0.05	7.93
	5: signal quality	0.32	1.55	1.72	0.10	2.37
	6: derivative	5.92	2.16	1.11	1.49	1.43
	7: squaring	5.94	2.18	7.66	4.46	5.78
	8: ABP beat quality	9.02	<b>32.87</b>	<b>16.90</b>	<b>12.88</b>	<b>15.13</b>
	9: ECG beat quality	<b>25.78</b>	<b>13.72</b>	<b>21.46</b>	<b>30.23</b>	<b>19.55</b>
	10: heart rate fusion	0.89	4.64	<b>14.78</b>	0.25	<b>16.98</b>
Memory arbiter		-	1.75	0.97	1.73	0.59
MIPS		2.78	3.79	5.74	5.45	6.58

Note: the percentages are for the processing logic part (RRHMS hardware except memory: FU ASIC accelerator and MIPS controller).

or derived from Table 6.2, Table 6.3, and Figure 6.3. The highest three percentages of each breakdown result in the table columns (Table 6.4) are highlighted.

The runtime breakdowns of the FPGA and ASIC implementations are the same, as they are implemented with the same hardware design. In the current prototype, the FUs are executed one by one (an FU starts execution after the previous one finishes). So there is no overlap in the runtime of the FUs. The runtime breakdown of the memory arbiter is not profiled because its operation is in parallel with the FU executions. From the runtime breakdown, we can see that the FUs are responsible for most (97.22%) of the computations. This is expected, as the FUs are designed to support the major processing steps of the RRHMS heart rate detection application, and the MIPS controller is mainly responsible for FU configurations and the scheduling of FU executions. Among the FUs, FU0 (low-pass filter), FU2 (peak detection), and FU9 (ECG beat quality) have the highest runtime percentages. Both low-pass filter and ECG beat quality (kurtosis computation) are

computation intensive, as they have to process a large amount of input data. Peak detection is control intensive due to the threshold-based detection rules (described by Algorithm 1 in Section 3.2).

The breakdowns of the resource utilization and power consumption are different in the FPGA and ASIC implementations. The resource breakdown of FPGA is based on the number of look-up tables (LUTs) used to implement the corresponding module. The ASIC resource breakdown is calculated using the area of each module. Most of the hardware resources are used for FU implementations in both platforms (94.46% in FPGA and 93.28% in ASIC). Therefore, although the proposed fault tolerance mechanism only protects the FUs, most parts of the RRHMS hardware system are covered as discussed in Section 5.2. Additionally, the total resource utilizations of the FU modules shared between ABP and ECG processing (as highlighted in Figure 3.1) are 43.53% and 46.16% in FPGA and ASIC, respectively. This proves the effectiveness of the algorithmic optimizations for sharing between ABP and ECG processing, without which those resources might be double what they are. The difference between the resource breakdown in the two implementations is due to differences in the ways their hardware logics are synthesized and implemented. In the ideal case, more hardware resources should be allocated to the module that has the higher computation utilization (higher runtime percentage). This is not the case in either FPGA or ASIC because some FUs (processing steps) are not mathematically complicated but their input data size and computation iteration number are large. FU0 (low-pass filter) and FU1 (slope sum) are examples.

The power breakdowns for FPGA and ASIC implementations are directly obtained from the power report generated by the Xilinx Power Analyzer and Synopsys Design Compiler, respectively. Similar to resource utilization, most of the hardware power is consumed in the FUs (92.82% in FPGA and 92.83% in ASIC). This makes sense because most of the computations are completed by the FUs. In general, the modules that have higher runtime percentages consume more power. However, there are exceptions. For example, the runtime percentage of FU2 (peak detection) is high (24.43%), but it is not among the highest three power consuming modules. This is because its execution contains several serialized threshold-based steps, and in each step, only a small subset of its logics is active. On the other hand, FU10 (heart rate fusion) has a very small runtime percentage (0.89%), but

in ASIC implementation, it both has high resource utilization (14.78%) and consumes a large amount of power (16.98%). No clear reason for this is found by the initial inspection in the ASIC code; further investigation is needed to explain it. The breakdown results provide guidance to further optimize the RRHMS design, such as hardware resource allocation and software/hardware partitioning (Section 7.2).

## 6.4 Fault Tolerance Evaluation

The proposed fault tolerance mechanism with the hardware fault detection and recovery unit (FDRU) is evaluated in this section. First, the overheads of FDRU in resource, power consumption, and performance are discussed, followed by an introduction to the fault injection method used. The fault tolerance coverage of FDRU is then analyzed by comparing the correctness of heart rate detection under injected faults when FDRU is and is not applied. Finally, the fault tolerance coverage is discussed, including the relationship between the amount of data used for result invariant profiling and FDRU coverage.

### 6.4.1 FDRU Overheads

As introduced in Chapter 5, the proposed fault tolerance mechanism is implemented in FDRU. The overheads of FDRU in resource utilization, power consumption, and performance are listed in Table 6.5. For resource overhead, the FDRU implementation needs an extra 14.54% of the look-up tables used by the baseline RRHMS hardware system on the FPGA platform. In ASIC, FDRU needs an extra 15.54% of cell gates to implement, which accounts for a 12.65% increase in die area. The resource overhead is due to:

- (1) FDRU’s controlling state machine logics (illustrated by Figure 5.3),
- (2) all FU configuration registers (listed in Table 4.1),
- (3) timeout registers for each FU and a heartbeat counter (16-bit each),  
and
- (4) address and result invariants of each FU, as well as invariant checkers for each invariant (hardware comparators).

Table 6.5: FDRU overheads in FPGA and ASIC implementations

	<b>FPGA</b>	<b>ASIC</b>
<b>Resource</b>	14.54% look-up tables	15.54% cell gates (12.65% area)
<b>Power*</b>	37.01%	33.89%
<b>Performance*</b>	0%	

\*Power and performance overheads in the table are the overheads during normal monitoring when no fault is detected. If there are faults, more overhead is introduced by fault recovery (FU reconfiguration and re-execution).

During normal monitoring, when there is no fault, FDRU consumes 37.01% and 33.89% more power in FPGA and ASIC implementations, respectively. The extra power consumption is due mainly to the invariant and heartbeat checking for FU fault detection. In addition, FDRU does not introduce any performance overhead during normal monitoring because all the fault detection checking is executed in parallel with FU executions. As a result, FDRU does not incur extra cycles during the normal FU execution. Besides, none of the fault detection checking is on the critical path of the RRHMS hardware system, and therefore the clock frequency is not affected by FDRU either. When faults are detected during the FU execution, extra power and performance overheads are introduced by the fault recovery process, as FDRU needs to reconfigure and re-execute the faulty FU to recover from the fault. The specific numbers of the extra overheads depend on two factors:

- (1) The frequency of FU fault detection. This is affected by the manifested FU fault rate. When an FU fault is detected, the FU is reconfigured and re-executed by FDRU, which consumes extra power.
- (2) The specific FU involved. Different FUs have different numbers of configuration registers and are responsible for different computations, so the performance and power overheads incurred by them vary.

If the two factors are known (by simulation or in real-life testing), the specific extra overheads can be calculated with the combined information of the FU runtime and power breakdowns (Table 6.4).

Compared with the traditional fault tolerance technique of double modular redundancy (DMR), where each FU must be duplicated, FDRU introduces much smaller resource and power overheads. The resource and power overheads in DMR are over 100% for each, as DMR requires duplicate modules

and operations for fault detection. The performance overheads during normal execution without fault are the same (0%) in DMR and FDRU. When a fault occurs in DMR, only the transient fault is recoverable through re-execution, as in the proposed FDRU. However, the tradeoff for the small overheads of FDRU is lower fault detection coverage. DMR can detect any fault that causes a result mismatch in the duplicate executions, while FDRU is only able to detect a fault that results in an FU hang or invariant violation. Detailed discussions of FDRU’s fault tolerance coverage are provided in the following sections.

### 6.4.2 Fault Injection Methodology

To simulate the behavior of low-level hardware faults with high fidelity, we applied hardware gate-level fault injection, as discussed in Section 5.1. We modified the CrashTest fault injection framework [107] to inject the gate-level fault to the RRHMS hardware system.

Figure 6.4 outlines the simulation flow for the fault injection experiment. The flow starts with the synthesis of the hardware design written in hardware description language (HDL). For RRHMS, the hardware is written in both VHDL and Verilog. GTECH library (a technology-independent library) is used for hardware synthesis. The Synopsys Design Compiler is used for synthesis of fault injection as well, but with a different technology library. After synthesis, a netlist file is generated with GTECH gate cells that implement the RRHMS hardware logics.

The netlist file is then input to the Perl script in the CrashTest framework to inject faults into the logic gates. Since the name of each GTECH cell gate

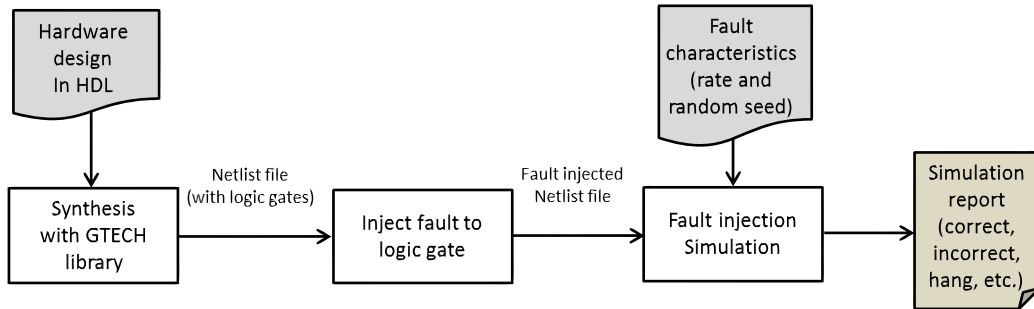


Figure 6.4: Fault injection simulation flow

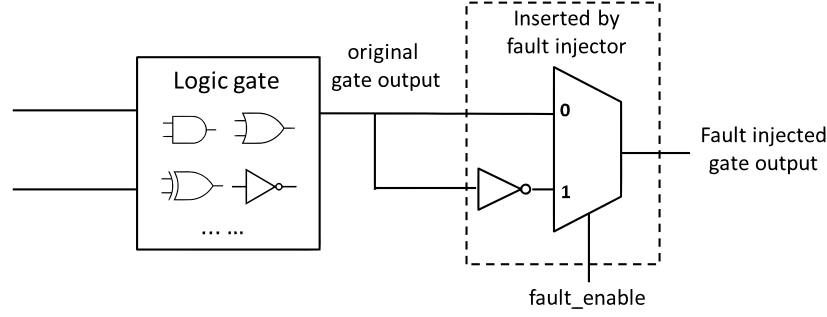


Figure 6.5: Fault injection to logic gate

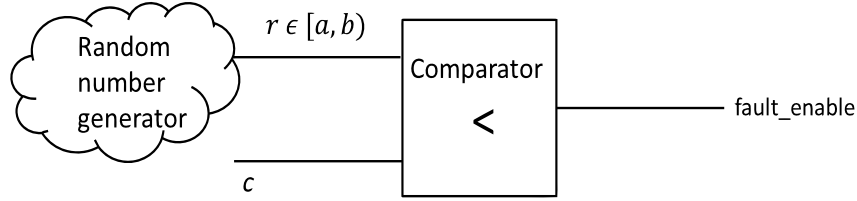


Figure 6.6: Fault enable generation

starts with *GTECH* (such as *GTECH\_AND* and *GTECH\_NOT*), it is easy for the Perl script to find all the logic gates. In *CrashTest*, the number of logic gates into which to inject faults can be specified by the user, and then the Perl script randomly selects the gate for fault injection. Since *RRHMS* hardware is not very big (around 43,000 *GTECH* gates for fault injection), we inject the fault into every logic gate for high fidelity fault injection simulation, as a fault may happen to any gate in real life. Figure 6.5 illustrates how the fault is injected by the Perl script. A 2-to-1 MUX is inserted into the output of the logic gate. One of the inputs of the MUX is directly connected with the original output of the logic gate, and the other input is connected to the inversion of the original output. So the selection signal of the MUX is used as the fault enable signal. When the fault enable signal is 0, the fault-injected gate outputs its original result (as if no fault has occurred). Otherwise, it outputs its inversion (as if a fault has occurred and flipped its output). All the fault enable signals are ported as the inputs of the top-level module, and the fault of each injected logic gate can be disabled or triggered in any cycle during the simulation by setting the corresponding fault enable signal.

After fault injection, the faulty behavior of the hardware can be simulated with the fault-injected netlist file. We use *Modelsim* for cycle-accurate simulation instead of applying the FPGA simulation as proposed in the *CrashTest*

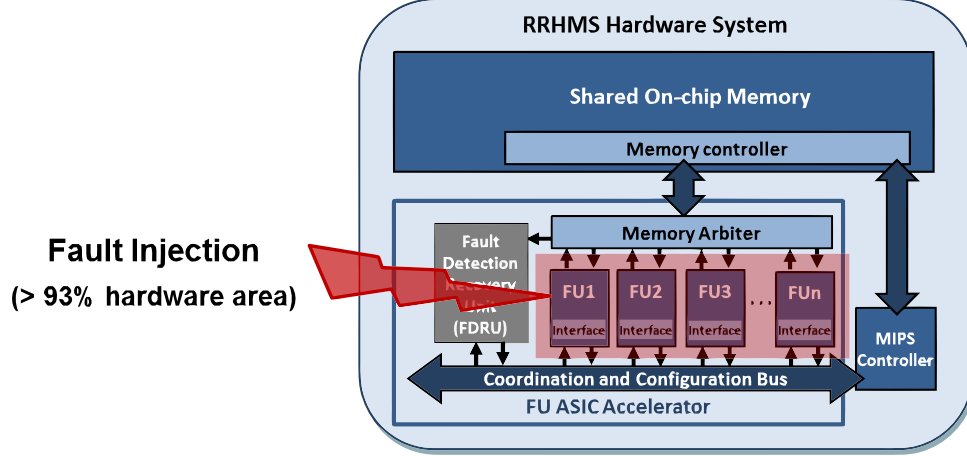


Figure 6.7: Fault injection area of the RRHMS hardware system

framework. This is done because FPGA does not have enough programmable ports to control all the fault enable signals (around 43,000) during RRHMS simulation. Since we focus on transient fault detection and recovery, only transient faults are injected, by transiently setting the fault enable signal to 1. This is realized using a random number generator and comparator as depicted in Figure 6.6. In each simulation cycle, the random number generator generates an integer value  $r$  uniformly distributed between  $a$  (inclusive) and  $b$  (exclusive). This value is compared with a pre-set value  $c$  by the comparator. If  $r < c$ , the fault enable signal is set to 1, otherwise to 0. Therefore the simulated transient fault rate is

$$\text{simulated transient fault rate} = \frac{c - a}{b - a}$$

In the RRHMS fault injection simulation, each fault enable signal is connected to a separate set of the fault generation system in Figure 6.6. So the transient faults are injected into every logic gate independently. With each pre-set fault rate ( $c$  value), the RRHMS is simulated 500 times with different random seeds set in each simulation to collect the statistics of the simulation results. The next section discusses the simulation results in detail.

### 6.4.3 Fault Tolerance Coverage

To evaluate the coverage of the proposed FDRU, we inject transient faults at different rates into the FU area using the fault injection framework in-

troduced in the previous section. Figure 6.7 depicts the fault injection area of the RRHMS hardware system. Since FDRU only protects FUs, only the logic gates of FUs are injected. This accounts for more than 93% of the processing logics in RRHMS (FU ASIC accelerator and MIPS controller). For comparison, faults are injected into both the baseline hardware system without FDRU and the fault tolerant hardware system with FDRU.

The comparison results are shown in Figure 6.8. The results are collected by injecting transient faults with six fault rates, from  $4 \times 10^{-9}$  to  $24 \times 10^{-9}$ . The fault rate here indicates how frequently the output of a logic gate is flipped (by setting the corresponding fault enable signal to 1). For example, the fault rate of  $4 \times 10^{-9}$  means that each faulty logic gate, on average, is injected with four faults in every  $10^9$  cycles of the hardware execution, where a fault is injected by flipping the output of a logic gate for one cycle. This is realized by setting the values of  $a$ ,  $b$ , and  $c$  in Figure 6.6 to 0,  $10^9$ , and 4, respectively. It should be noted that the fault rate is for each single logic gate and that the faults are injected independently for each gate. Therefore, in RRHMS fault injection, since the FUs are synthesized to around 43,000 logic gates, at the fault rate of  $4 \times 10^{-9}$ , a total of  $43,000 \times 4 = 172,000$  faults are injected into the FUs on average in every  $10^9$  cycles of hardware execution. Faults may be injected to an FU during its re-execution for recovery from previously detected faults. When this happens, the new faults can be detected in the same way by FDRU during the re-execution, as FDRU handles a re-execution in exactly the same way it handles a normal execution (as discussed in Section 5.4).

Table 6.6: Output of the RRHMS in fault injection simulation

	Baseline System	Fault Tolerance System (with FDRU)
<b>Correct Result</b>	The hardware finishes execution in time, and the heart rate detected is correct.	
<b>Incorrect Result</b>	The hardware finishes execution in time, but the heart rate detected is incorrect.	
<b>System Failure</b>	The hardware does not finish execution in time.	Either FU hang or invariant vio- lation is detected three times in a single MIPS-scheduled execu- tion of the FU.

Note: In time means within five times of the supposed execution time (obtained by the same application simulation without fault injection).



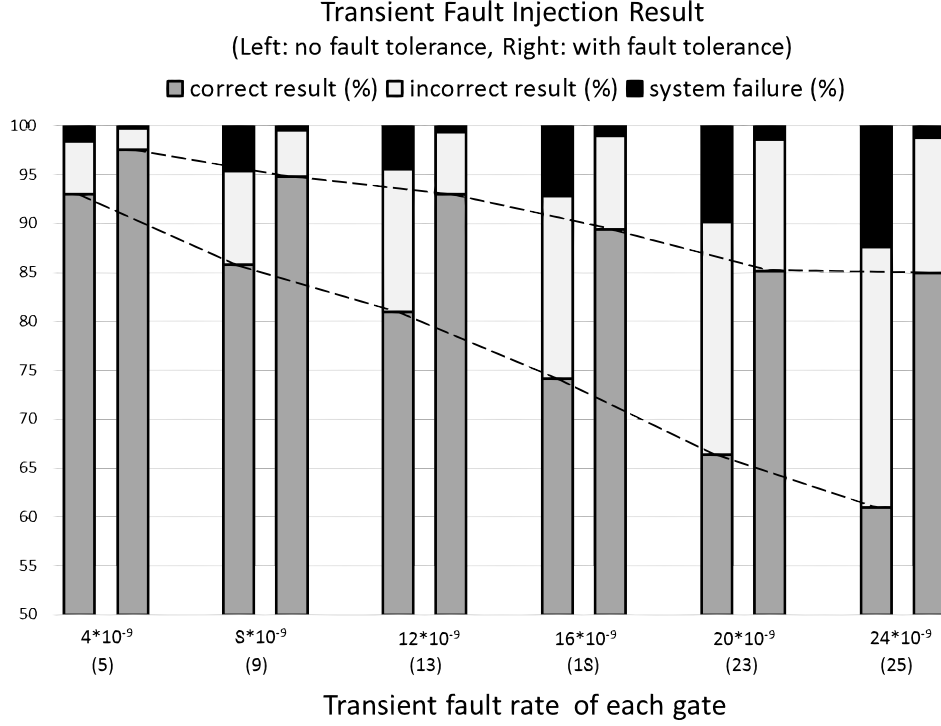


Figure 6.8: Fault injection results comparison between baseline and fault tolerance hardware systems (result invariants are profiled with 1000 windows of patient data (a40050 of the MIMIC II database))

For each fault rate, RRHMS is simulated 500 times with different random seeds set for the random number generators used to enable faults (Figure 6.6). The number in parentheses in Figure 6.8 below the fault rate is the average number of faults triggered (enabled) in the 500 simulations under the corresponding fault injection rate. The application that runs on the fault-injected RRHMS hardware is the RRHMS heart rate detection application developed in this thesis (Chapter 3). Only one window of the patient data (patient a40050 from the MIMIC II database) is used in the fault injection simulation to save simulation time. This is because post-synthesis simulation with 43,000 random number generations in every cycle to control the injected faults is slow. So the output of the application is the heart rate detected from the window by the analysis of both ABP and ECG signals. Since the application is short (only processing one window of patient data), the fault rate is set to be high to magnify the effects of the fault. For the fault tolerance system, the result invariants are profiled using 1000 windows of patient data (the result invariants of all FUs are listed in Table 5.2).

There are three possible outputs (as listed in Table 6.6) for the fault injection simulation: *correct result*, *incorrect result*, or *system failure*. In both baseline and fault tolerance hardware systems, the correct result indicates that the hardware system finishes the execution within five times the supposed execution time and that the detected heart rate of the window is correct (every output bit matches the golden result). The supposed execution time and golden result are obtained by simulating the same application without fault injection. The correct result happens when the injected fault does not manifest or propagate to affect the application result. This is possible in these cases:

- (1) The fault is injected into a logic gate that is not in use in the cycle during which its output is flipped by the injected fault.
- (2) The fault affects some intermediate results but does not propagate to the final result. For example, if the fault changes the quality of one ABP beat from 0.9 to 0.6, the beat is still counted as good and does not affect the ABP signal quality of the window; therefore, it does not affect the final heart rate estimation.
- (3) The injected fault is detected and fixed through re-executions in the case of the fault tolerance system with FDRU.

On the other hand, the incorrect result in both systems means that the hardware execution finishes within five times the supposed execution time, but the detected heart rate is incorrect (at least one of the output bits does not match the golden result). System failure has different meanings in the two systems. In the baseline system, system failure means the system does not finish the execution within five times the supposed execution time, which is an indicator of FU hang. FU hang is usually caused by one of two scenarios:

- (1) The fault is injected into the FU state controller and results in infinite FU state loops.
- (2) The fault corrupts an intermediate value used for FU control logics, such as the loop iteration variable.

In the fault tolerance system, system failure indicates that either an FU hang or an invariant violation is detected three times in an MIPS-scheduled execution of the FU (FDRU enters *State 5* in Figure 5.3). The fault tolerance

hardware system never fails by exceeding five times the supposed execution time. This is because FDRU would have detected the corresponding FU hang before that happens.

As shown in Figure 6.8, for both baseline and fault tolerance systems, as the fault rate increases, the percentages of the correct results in 500 simulations (with different random fault injections) decrease and the percentages of incorrect results increase. This is because with the higher fault rate, the probability that the injected faults will impact the critical gates is higher. The proposed fault tolerance mechanism is able to increase the correct result percentages under each fault injection rate by detecting the FU fault and recovering from it through re-executions. A large number of both incorrect results and system failures (FU hangs) in the baseline system are detected and recovered by FDRU. System failures are almost eliminated in the fault tolerance system due to the FU hang detection with heartbeat, and the total percentages of incorrect system behaviors (incorrect results and system failures) are reduced by at least 55.95% in all fault injection rates (e.g., reduced from 7% to 2.4% under the  $4 \times 10^{-9}$  fault rate). So at the fault rate of  $24 \times 10^{-9}$ , the percentage of correct results increases from 61.00% to 85.00%. In addition, it should be noted that system failures are detected in the fault tolerance system. Therefore, the user can be notified of the failure, even though the baseline system cannot detect it. However, there are still incorrect results in the fault tolerance system, since some faults may cause incorrect results that are within the variant range and therefore not detectable by the invariant checking of FDRU. This is the tradeoff of the FDRU's low area (resource) and power overheads compared with the DMR mechanism. DMR may still have incorrect results if the faults cause the duplicated modules to give the same incorrect results or occur to the DMR voters, but the percentage is expected to be smaller. Implementing DMR to protect FUs for comparison with FDRU is part of our future work.

#### 6.4.4 Discussion of Fault Coverage

As discussed in Section 5.5, the fault detection coverage of the proposed FDRU mechanism depends on result invariant profiling. If the profiling data size is small, the profiled result invariant range is small, and false positives

may occur if a new input data scenario appears. On the other hand, if the profiling data size is large, the profiled result invariant range is likewise large, and as a result, faults that cause a result that is incorrect but not out of range cannot be detected. This section discusses the effect of the profiling data size on fault tolerance coverage.

The fault tolerance results shown in Figure 6.8 are obtained by profiling the result invariant with 1000 windows of patient (a40050) data, which are used to run the heart rate detection application for one-window heart rate detection. The result invariants of all FUs profiled by 1000 windows are listed in Table 5.2. To compare fault coverage when different profiling data sizes are used, the result invariants are also profiled with one window (just the window used in fault injection simulations) and with 4000 windows (11.11 hours of the patient data that includes 11 arrhythmia problems). Table 6.7 and Table 6.8 show the result invariants of all FUs profiled with 1 and 4000 windows, respectively. Comparing the three tables (Table 5.2, Table 6.7, and Table 6.8), we can see that the range of each result invariant increases (the lower bound becomes smaller and the upper bound becomes larger) as the

Table 6.7: FU result invariants, profiled with one window (10 s) of patient data (a40050) from the MIMIC II database

Functional Unit		Result Invariant			
		<i>min</i>	<i>max</i>	<i>min<sub>diff</sub></i>	<i>max<sub>diff</sub></i>
FU0 - low-pass	for ABP	49.20	97.20	-3.00	7.20
	for ECG	-5.24	3.60	-2.03	0.81
FU1 - slope sum	for ABP	0.00	45.00	-7.20	7.20
	for ECG	0.00	4.74	-2.75	2.75
FU2 - peak detection	peak index	44.00	1178	80.00	82.00
	peak number	14.00	15.00	-	-
FU3 - heart rate		92.51	92.59	-	-
FU4 - Kalman filter	filtered value	90.23	90.24	-	-
	residue	0.23	0.24	-	-
FU5 - signal quality		1.00	1.00	-	-
FU6 - derivative		-2.03	0.81	-1.10	1.14
FU7 - squaring		0.00	4.12	-2.82	2.75
FU8 - ABP beat quality		0.99	1.00	-	-
FU9 - ECG beat quality		4.64	5.45	-	-
FU10 - heart rate fusion		90.23	90.23	-	-

Note: the following conditions hold true during the corresponding FU execution:  $y_i \geq min$ ,  $y_i \leq max$ ,  $y_i - y_{i-1} \geq min_{diff}$ , and  $y_i - y_{i-1} \leq max_{diff}$ , where  $y_i$  is the current result value and  $y_{i-1}$  is the previous result value.

Table 6.8: FU result invariants, profiled with 4000 windows (11.1 hours) of patient data (a40050) from the MIMIC II database, during which there are 11 doctor annotated arrhythmia problems

Functional Unit		Result Invariant			
		<i>min</i>	<i>max</i>	<i>min<sub>diff</sub></i>	<i>max<sub>diff</sub></i>
FU0 - low-pass	for ABP	30.00	176.10	-26.10	26.40
	for ECG	-9.05	8.76	-4.16	4.88
FU1 - slope sum	for ABP	0.00	110.40	-26.40	67.20
	for ECG	0.00	24.07	-18.05	24.07
FU2 - peak detection	peak index	0.00	1250.00	37.00	354.00
	peak number	1.00	28.00	-18.00	21.00
FU3 - heart rate		53.35	175.52	-58.43	64.31
FU4 - Kalman filter	filtered value	11.58	160.10	-35.68	22.56
	residue	-132.05	83.49	-	-
FU5 - signal quality		0.00	1.00	-	-
FU6 - derivative		-4.16	4.88	-8.46	8.18
FU7 - squaring		0.00	23.81	-14.75	18.05
FU8 - ABP beat quality		0.00	1.00	-	-
FU9 - ECG beat quality		1.00	34.09	-	-
FU10 - heart rate fusion		71.19	160.10	-42.25	16.98

Note: the following conditions hold true during the corresponding FU execution:  $y_i \geq min$ ,  $y_i \leq max$ ,  $y_i - y_{i-1} \geq min_{diff}$ , and  $y_i - y_{i-1} \leq max_{diff}$ , where  $y_i$  is the current result value and  $y_{i-1}$  is the previous result value.

profiling data size increases. There are a few extra missing invariant values in Table 6.7 compared with the other two tables. This is because only a single result is calculated from a window for some invariant variables (such as peak number and signal quality), so just profiling one window cannot obtain the consecutive result difference of those variables. Therefore, those invariants are ignored in the simulations using invariants profiled with one window.

The invariants profiled with one and 4000 windows are used to run the same application with the same input data as with 1000 windows for fault injection simulations. Figure 6.9 illustrates the comparisons of the transient fault injection results between the baseline system and the fault tolerance systems with different profiling data sizes. The result bars of each fault injection rate are (from left to right): baseline system, fault tolerance system profiled with one window, fault tolerance system profiled 1000 windows, and fault tolerance system profiled with 4000 windows. The three outputs (correct result, incorrect result, and system failure) are the same as used in Figure 6.8 and explained in Table 6.6. Similarly, each fault rate is simulated 500

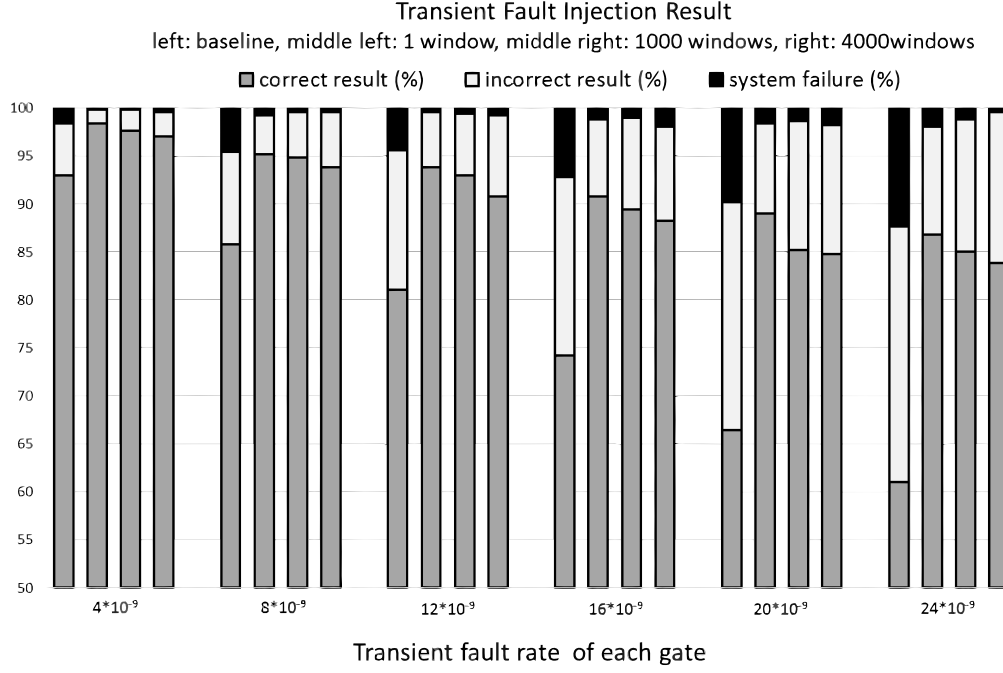


Figure 6.9: Fault injection result comparison between baseline and fault tolerance hardware systems with different profiled data sizes

times in each system with different fault injection random seeds.

As expected, fault tolerance coverage decreases as the profiling data size increases. This is because a larger invariant range fails to detect faults that cause the result to be incorrect but still within the range. However, the coverage difference between the fault tolerance systems with different profiling data size is small. This is because in most cases of incorrect results caused by hardware faults the result is greatly changed by the fault and goes out of the invariant range by a large amount. Therefore, even though the invariant ranges are increased with more profiling data, they are still effective in detecting most of the faults that can be detected with less profiling data. With the profiling data size of 4000 windows, which includes 11.11 hours of the patient data and 11 arrhythmia occurrences, FDRU is still able to reduce the incorrect system behaviors (incorrect results and system failures) by at least 51.58% compared to the baseline system. For example, the total percentage of incorrect results and system failures at the fault rate of  $12 \times 10^{-9}$  is reduced from 19% to 9.2%). So, result invariants should be profiled with long periods of patient data to reduce false positives. It is fine to set the result invariants based on the patient's physiological limitations, as increasing

the result invariant range within the patient’s physiological range does not decrease the fault coverage by much.

Another interesting result is that as profiling data size increases, the system failure percentage decreases (e.g., when the fault rate is  $8 \times 10^{-9}$  and  $24 \times 10^{-9}$  in Figure 6.9). This is because as the invariant range becomes larger, the probability of three result invariant violations in a single MIPS scheduled FU execution becomes smaller. Therefore, some of the previous system failures change to incorrect results.

The fault injection results presented and discussed here are obtained with the data of only one patient (patient a40050 from the MIMIC II database). This is done because of the long fault injection simulation time; it takes about a week to simulate with one patient’s data on a single computer. For the monitoring of other patients, the result invariants need to be profiled with the new patient data, as the physiological ranges, such as the blood pressure range, may vary for different patients. Fault coverage results similar to those in Figure 6.8 and Figure 6.9 are expected when other patient data is used, because even though the physiological ranges of different patients differ, the faulty behavior of the hardware system is similar. Most faults would result in FU hangs and incorrect results that are far beyond the physiological ranges and which can be easily detected and recovered from using the proposed FDRU scheme.

## Chapter 7

# CONCLUSION AND FUTURE WORK

This chapter concludes the thesis work introduced in the previous chapters and discusses future work to further improve the efficiency and fault tolerance of the proposed RRHMS.

### 7.1 Conclusion

This thesis presents the robust and reliable heart rate monitoring system (RRHMS). RRHMS provides accurate, portable, and long battery life for heart rate monitoring in real time. It is robust because it estimates the heart rate from both ABP and ECG signals and fuses the results according to their signal quality. It is reliable because it applies a low-overhead fault tolerance mechanism in the underlying hardware to provide reliable processing for heart rate detection even with hardware faults. Design and optimization in both algorithm and hardware have been considered in developing RRHMS. Its heart rate detection algorithm is developed by applying algorithmic optimizations to merge the separately developed ABP and ECG beat detection steps into shared steps. The shared steps enable shared hardware modules for efficient and portable RRHMS hardware design. The algorithmically optimized ABP and ECG beat detection can match the results of the original algorithms, and the heart rate fusion applied provides continuous heart rate monitoring when a single signal is corrupted. To efficiently support the processing of ABP and ECG signals, an embedded hardware framework with configurable functional units (FUs) is proposed. FUs are ASIC accelerators that support each processing step and can be quickly configured for ABP and ECG computations as well as for patient-specific monitoring. The proposed RRHMS hardware system is implemented both on ASIC and FPGA platforms. Compared with the Android implementation that runs on the



Qualcomm Krait processor, both ASIC and FPGA implementations achieve better runtime performance (20.6 and 13.7 time speedups to Android, respectively) with lower power consumption ( $1/139$  and  $1/67$  of Android’s power, respectively). In the end, a fault detection and recovery unit (FDRU) is proposed to provide low-overhead fault tolerance in FUs by applying invariant checking and heartbeats. Both transient and permanent faults can be detected by FDRU, but only transient faults can be recovered from. FDRU improves the reliability of the RRHMS hardware system by increasing correct output percentages under injected transient hardware faults. It is also able to reduce 55.95% of incorrect output and system failure cases at all tested fault rates while incurring only about a 15% area (resource) overhead in the hardware logics. During normal monitoring without hardware faults, FDRU incurs only a 34% power overhead (due to fault detection checking) and introduces no performance overhead.

## 7.2 Future Work

This section discusses the future directions we plan to explore to further optimize the performance, resource utilization, and power efficiencies of RRHMS, as well as to improve the RRHMS fault tolerance capability at with minimum overhead.

### 7.2.1 Software/Hardware Partitioning

Even though the proposed RRHMS hardware system achieves much better performance and power efficiency than the Android implementation on the Qualcomm Krait processor, the runtime, resource, and power breakdowns of the RRHMS hardware modules (in Table 6.4) may not achieve the best efficiency. For example, FU8 (ABP beat quality) accounts for a large percentage of resource utilization (32.87% in FPGA and 16.90% in ASIC), but its runtime percentage (9.02%) is high. This means that among the hardware modules, FU8 spends a large amount of the resource budget, but the amount of work it completes does not seem to justify all the resources it takes up. Similarly, FU4 accounts for 6.34% and 9.77% of the resources in FPGA and ASIC, respectively, but only contributes 0.29% of the application cycles. On

the other hand, FU0 (low-pass filter) accounts for 11.88% of the application runtime, while it takes up only 4.08% and 3.76% of the FPGA and ASIC resources, respectively. Therefore, the current hardware modules may not achieve the best efficiency in balancing the tradeoffs among performance, resource utilization, and power consumption. For instance, since FU4 only accounts for 0.29% of the runtime while using a relatively large portion of the resource, it may be better to put its execution in the MIPS controller as the software program. As a result, even though MIPS is not as efficient as FU4 in supporting its execution (e.g., MIPS may take five times FU4’s execution time and expend more energy than FU4), the resources used by FU4 (6.34% in FPGA and 9.77% in ASIC) can be saved. This boils down to trading performance for resources (or hardware area), so the question arises: what is the best tradeoff, the best software/hardware partitioning point? To answer this question, a metric needs to be developed that puts different weights on performance, resource utilization, and power consumption, and then these weights need to be set according to the specific application requirements. With such a metric, we can do more numeric analysis and experiments to explore the design space to optimize the RRHMS design for efficiency.

## 7.2.2 Fault Behavior Analysis

The proposed fault detection and recovery mechanism of FDRU is effective in reducing incorrect system behavior (incorrect results and system failures) by 55.95%. However, there is still room for improvement, as the injected faults still cause a lot of incorrect system behavior in the fault tolerance system, especially when the fault rate is high (15% at fault rate of  $24 \times 10^{-9}$ ). Therefore, further analysis of FU fault behavior will help to improve fault coverage at low overhead. We plan to do the following:

- (1) Analyze which FU and which part of the FU is most susceptible to hardware faults.
- (2) Analyze how faults propagate to affect the application result and cause the system to behave incorrectly.

Since the FUs are built to support the specific heart rate monitoring application, FU behavior, such as execution and output patterns, are application-specific. Therefore, FU fault behaviors should be application-specific as well,

and patterns may be found. The above two analyses are useful in finding such patterns.

Understanding which FUs are more susceptible to hardware faults helps us to better provide fault tolerance and allocate hardware resource. To gain this understanding, we can inject faults at the same rate to each FU separately and collect the output statistics. If an FU causes much higher percentages of incorrect results and system failures, it needs more fault tolerance support. On the other hand, if an FU causes much lower percentages of incorrect results and system failures, its fault tolerance support can be reduced. Intuitively, FU8 (ABP beat quality) and FU9 (ECG beat quality) are less susceptible to hardware faults, because a beat is classified as good as long as its quality is above a certain threshold. So for example, if the threshold for a good ABP beat is 0.5 and a beat's quality is supposed to be 0.9, then as long as the beat quality is computed to be larger than 0.5, the beat will be classified as good and not affect the final heart rate output. This will remain the case even if the fault causes the beat quality to be incorrectly computed as 0.6 or 100). If this intuition is true, less hardware can be allocated for the fault detection checking in FU8 and FU9. The saved hardware resource can be used to strengthen the protection for FUs that are more vulnerable to faults. As a result, with the same FDRU hardware resource budget, the system can be made more resilient to tolerate faults. Similarly, since the FUs are designed to be modular and follow the same design template, understanding which part of the FU (bus/memory interface, configuration register, computation data path, state machine controller) is more vulnerable to faults helps us to better design the FU template or to guide the fault tolerance FU design for using the proposed hardware system architecture as the framework.

Analyzing the fault propagation path helps us find the data paths that are most susceptible to faults so that we can select the most vulnerable ones to protect without introducing much overhead. For example, if a data path segment is on all the fault propagation paths that lead to incorrect outputs, protecting this segment does not incur much overhead and can improve the system fault tolerance capability. Analysis of fault propagation is difficult, however. It requires back-tracing from the final incorrect output to the source of the fault, and this process needs to be repeated with a large number of simulations. As a result, the statistics of the data path segments can be generated to show which segments have appeared on the most paths that

lead to incorrect outputs.

### 7.2.3 Functional Units Pipelining

Currently, the FUs are scheduled by the MIPS controller to be executed one by one. An FU is scheduled for execution only when the previous FU's execution finishes. Even though the current design is already fast enough to meet the real-time constraint, pipelining FU executions may gain further runtime efficiency and allow aggressive frequency and voltage scaling to save total energy consumption and achieve longer battery life. But this is currently just speculation and needs more analysis or experiments to verify. Pipelining FU executions would introduce extra hardware complexity, and as a result, the energy saving through performance gain may be paid for by the energy consumed in the extra hardware logics. Although the current hardware already has some features to support parallel FU executions, such as the design of the memory arbiter and bus arbiter, the following modifications are still needed for FU pipelining:

- (1) Modification of the FU execution instruction to include information on FU dependency to prevent RAW (read-after-write) hazard.
- (2) Maintenance of a table in the MEM stage of the MIPS controller to keep track of which FUs are executing and which are idle.
- (3) Modification of FDRU to allow parallel FU invariant and heartbeat checking to support parallel fault tolerance in FU executions.

## References

- [1] “Heartcheck ECG monitor website,” Oct. 2014. [Online]. Available: <http://www.theheartcheck.com/>
- [2] “Wikipedia. Cardiac dysrhythmia,” Mar. 2014. [Online]. Available: [http://en.wikipedia.org/wiki/Cardiac\\_dysrhythmia/](http://en.wikipedia.org/wiki/Cardiac_dysrhythmia/)
- [3] H. Alemzadeh, C. Di Martino, Z. Jin, Z. T. Kalbarczyk, and R. K. Iyer, “Towards resiliency in embedded medical monitoring devices,” in *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*. IEEE, 2012, pp. 1–6.
- [4] J. Allen and A. Murray, “Assessing ECG signal quality on a coronary care unit,” *Physiological Measurement*, vol. 17, no. 4, p. 249, 1996.
- [5] S. T. Lawless, “Crying wolf: False alarms in a pediatric intensive care unit,” *Critical Care Medicine*, vol. 22, no. 6, pp. 981–985, 1994.
- [6] C. L. Tsien and J. C. Fackler, “Poor prognosis for existing monitors in the intensive care unit,” *Critical Care Medicine*, vol. 25, no. 4, pp. 614–619, 1997.
- [7] M.-C. Chambrin, P. Ravau, D. Calvelo-Aros, A. Jaborska, C. Chopin, and B. Boniface, “Multicentric study of monitoring alarms in the adult intensive care unit (ICU): A descriptive analysis,” *Intensive Care Medicine*, vol. 25, no. 12, pp. 1360–1366, 1999.
- [8] P. Lynn, “Online digital filters for biological signals: Some fast designs for a small computer,” *Medical and Biological Engineering and Computing*, vol. 15, no. 5, pp. 534–540, 1977.
- [9] J. Sun, A. Reisner, and R. Mark, “A signal abnormality index for arterial blood pressure waveforms,” in *Computers in Cardiology, 2006*. IEEE, 2006, pp. 13–16.
- [10] T. Berset, D. Geng, and I. Romero, “An optimized DSP implementation of adaptive filtering and ICA for motion artifact reduction in ambulatory ECG monitoring,” in *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*. IEEE, 2012, pp. 6496–6499.

- [11] M. H. Ebrahim, J. M. Feldman, and I. Bar-Kana, "A robust sensor fusion method for heart rate estimation," *Journal of Clinical Monitoring*, vol. 13, no. 6, pp. 385–393, 1997.
- [12] G. B. Moody and R. G. Mark, "A database to support development and evaluation of intelligent intensive care monitoring," in *Computers in Cardiology, 1996*. IEEE, 1996, pp. 657–660.
- [13] Z. Kalbarczyk, R. K. Iyer, G. L. Ries, J. U. Patel, M. S. Lee, and Y. Xiao, "Hierarchical simulation approach to accurate fault modeling for system dependability evaluation," *Software Engineering, IEEE Transactions on*, vol. 25, no. 5, pp. 619–632, 1999.
- [14] P. Hazucha, T. Karnik, J. Maiz, S. Walstra, B. Bloechel, J. Tschanz, G. Dermer, S. Harelend, P. Armstrong, and S. Borkar, "Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25- $\mu$ m to 90-nm generation," in *Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International*. IEEE, 2003, pp. 21–5.
- [15] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, 2005.
- [16] I. Al Khatib, F. Poletti, D. Bertozzi, L. Benini, M. Bechara, H. Khalifeh, A. Jantsch, and R. Nabiev, "A multiprocessor system-on-chip for real-time biomedical monitoring and analysis: Architectural design space exploration," in *Proceedings of the 43rd Annual Design Automation Conference*. ACM, 2006, pp. 125–130.
- [17] "Wikipedia. Krait CPU," Oct. 2014. [Online]. Available: [http://en.wikipedia.org/wiki/Krait\\_\(CPU\)/](http://en.wikipedia.org/wiki/Krait_(CPU))
- [18] J. Pan and W. J. Tompkins, "A real-time QRS detection algorithm," *Biomedical Engineering, IEEE Transactions on*, no. 3, pp. 230–236, 1985.
- [19] A. Pachauri and M. Bhuyan, "ABP peak detection using energy analysis technique," in *Multimedia, Signal Processing and Communication Technologies (IMPACT), 2011 International Conference on*. IEEE, 2011, pp. 36–39.
- [20] W. Zong, G. Moody, and R. Mark, "Reduction of false arterial blood pressure alarms using signal quality assesement and relationships between the electrocardiogram and arterial blood pressure," *Medical and Biological Engineering and Computing*, vol. 42, no. 5, pp. 698–706, 2004.

- [21] Q. Li, R. G. Mark, and G. D. Clifford, "Robust heart rate estimation from multiple asynchronous noisy sources using signal quality indices and a Kalman filter," *Physiological Measurement*, vol. 29, no. 1, p. 15, 2008.
- [22] A. Aboukhalil, L. Nielsen, M. Saeed, R. G. Mark, and G. D. Clifford, "Reducing false alarm rates for critical arrhythmias using the arterial blood pressure waveform," *Journal of Biomedical Informatics*, vol. 41, no. 3, pp. 442–451, 2008.
- [23] C. Pavlatos, A. Dimopoulos, G. Manis, and G. Papakonstantinou, "Hardware implementation of Pan & Tompkins QRS detection algorithm," in *Proceedings of the EMBEC05 Conference*, 2005.
- [24] C. I. Jeong, M. I. Vai, and P. U. Mak, "ECG QRS complex detection with programmable hardware," in *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE*. IEEE, 2008, pp. 2920–2923.
- [25] R. Stojanović, D. Karadaglić, M. Mirković, and D. Milošević, "A FPGA system for QRS complex detection based on integer wavelet transform," *Measurement Science Review*, vol. 11, no. 4, pp. 131–138, 2011.
- [26] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 99–123, 2001.
- [27] "Synopsys design compiler for RTL synthesis," Sep. 2014. [Online]. Available: <http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx>
- [28] W. Zong, T. Heldt, G. Moody, and R. Mark, "An open-source algorithm to detect onset of arterial blood pressure pulses," in *Computers in Cardiology, 2003*. IEEE, 2003, pp. 259–262.
- [29] "Important physiological signals in the body," May 2011. [Online]. Available: <http://biomedikal.in/2011/05/important-physiological-signals-in-the-body/>
- [30] P. Gamble, H. McManus, D. Jensen, and V. Froelicher, "A comparison of the standard 12-lead electrocardiogram to exercise electrode placements." *CHEST Journal*, vol. 85, no. 5, pp. 616–622, 1984.
- [31] E. H. Schwab and G. Herrmann, "Alterations of the electrocardiogram in diseases of the pericardium," *Archives of Internal Medicine*, vol. 55, no. 6, pp. 917–941, 1935.

- [32] H. Blackburn, A. Keys, E. Simonson, P. Rautaharju, and S. Punsar, "The electrocardiogram in population studies a classification system," *Circulation*, vol. 21, no. 6, pp. 1160–1175, 1960.
- [33] Y. Ohnishi, T. Inoue, and H. Fukuzaki, "Value of the signal-averaged electrocardiogram as a predictor of sudden death in myocardial infarction and dilated cardiomyopathy," *Japanese Circulation Journal*, vol. 54, no. 2, pp. 127–136, 1990.
- [34] D. Perloff, C. Grim, J. Flack, E. D. Frohlich, M. Hill, M. McDonald, and B. Z. Morgenstern, "Human blood pressure determination by sphygmomanometry." *Circulation*, vol. 88, no. 5, pp. 2460–2470, 1993.
- [35] J. Sun, A. Reisner, M. Saeed, and R. Mark, "Estimating cardiac output from arterial blood pressure waveforms: A critical evaluation using the mimic ii database," in *Computers in Cardiology, 2005*. IEEE, 2005, pp. 295–298.
- [36] K. Minolta, "Basic understanding of the pulse oximeter: How to read SpO<sub>2</sub>," 2006. [Online]. Available: <http://windward.hawaii.edu/facstaff/miliefsky-m/ZOOL%20142L/aboutPulseOximetry.pdf/>
- [37] A. Nimmo and G. Drummond, "Respiratory mechanics after abdominal surgery measured with continuous analysis of pressure, flow and volume signals." *British Journal of Anaesthesia*, vol. 77, no. 3, pp. 317–326, 1996.
- [38] L. Nilsson, A. Johansson, and S. Kalman, "Monitoring of respiratory rate in postoperative care using a new photoplethysmographic technique," *Journal of Clinical Monitoring and Computing*, vol. 16, no. 4, pp. 309–315, 2000.
- [39] J. F. Fieselman, M. S. Hendryx, C. M. Helms, and D. S. Wakefield, "Respiratory rate predicts cardiopulmonary arrest for internal medicine inpatients," *Journal of General Internal Medicine*, vol. 8, no. 7, pp. 354–360, 1993.
- [40] M. B. Simson, "Use of signals in the terminal QRS complex to identify patients with ventricular tachycardia after myocardial infarction." *Circulation*, vol. 64, no. 2, pp. 235–242, 1981.
- [41] M. K. Das, B. Khan, S. Jacob, A. Kumar, and J. Mahenthiran, "Significance of a fragmented QRS complex versus a Q wave in patients with coronary artery disease," *Circulation*, vol. 113, no. 21, pp. 2495–2501, 2006.
- [42] M. B. Messaoud, "Neuronal classification of atria fibrillation," *Leonardo Journal of Sciences*, vol. 12, pp. 196–213, 2008.



- [43] G. M. Friesen, T. C. Jannett, M. A. Jadallah, S. L. Yates, S. R. Quint, and H. T. Nagle, "A comparison of the noise sensitivity of nine QRS detection algorithms," *Biomedical Engineering, IEEE Transactions on*, vol. 37, no. 1, pp. 85–98, 1990.
- [44] M. S. Chavan, R. Aggarwala, and M. Uplane, "Suppression of baseline wander and power line interference in ECG using digital IIR filter," *International Journal of Circuits, Systems And Signal Processing*, vol. 2, no. 2, pp. 356–65, 2008.
- [45] P. S. Hamilton and W. J. Tompkins, "Quantitative investigation of QRS detection rules using the MIT/BIH arrhythmia database," *Biomedical Engineering, IEEE Transactions on*, no. 12, pp. 1157–1165, 1986.
- [46] M.-E. Nygård and J. Hulting, "An automated system for ECG monitoring," *Computers and Biomedical Research*, vol. 12, no. 2, pp. 181–202, 1979.
- [47] M. Okada, "A digital filter for the QRS complex detection," *Biomedical Engineering, IEEE Transactions on*, no. 12, pp. 700–703, 1979.
- [48] F. Gritzali, "Towards a generalized scheme for QRS detection in ECG waveforms," *Signal Processing*, vol. 15, no. 2, pp. 183–192, 1988.
- [49] W. P. Holsinger, K. M. Kempner, and M. H. Miller, "A QRS preprocessor based on digital differentiation," *Biomedical Engineering, IEEE Transactions on*, no. 3, pp. 212–217, 1971.
- [50] N. V. Thakor and Y. S. Zhu, "Applications of adaptive filtering to ECG analysis: Noise cancellation and arrhythmia detection," *Biomedical Engineering, IEEE Transactions on*, vol. 38, no. 8, pp. 785–794, 1991.
- [51] S. Kadambe, R. Murray, and G. F. Boudreaux-Bartels, "Wavelet transform-based QRS complex detector," *Biomedical Engineering, IEEE Transactions on*, vol. 46, no. 7, pp. 838–848, 1999.
- [52] S. Poornachandra, "Wavelet-based denoising using subband dependent threshold for ECG signals," *Digital Signal Processing*, vol. 18, no. 1, pp. 49–55, 2008.
- [53] J. Zaleski, "Signal artifact smoothing using the extended Kalman filter in real-time arterial blood pressure measurements," Oct. 2014. [Online]. Available: <http://www.medicinfotech.com/2014/10/signal-artifact-smoothing-realtime-arterial-blood-pressure-measurements/>

- [54] V. X. Afonso, W. J. Tompkins, T. Q. Nguyen, and S. Luo, "ECG beat detection using filter banks," *Biomedical Engineering, IEEE Transactions on*, vol. 46, no. 2, pp. 192–202, 1999.
- [55] C. Li, C. Zheng, and C. Tai, "Detection of ECG characteristic points using wavelet transforms," *Biomedical Engineering, IEEE Transactions on*, vol. 42, no. 1, pp. 21–28, 1995.
- [56] N. Thakor, J. Webster, and W. Tompkins, "Optimal QRS detector," *Medical and Biological Engineering and Computing*, vol. 21, no. 3, pp. 343–350, 1983.
- [57] I. I. Christov, "Real time electrocardiogram QRS detection using combined adaptive threshold," *BioMedical Engineering OnLine*, vol. 3, no. 1, p. 28, 2004.
- [58] P. Hamilton, "Open source ECG analysis," in *Computers in Cardiology, 2002*. IEEE, 2002, pp. 101–104.
- [59] M. Aboy, J. McNames, T. Thong, D. Tsunami, M. S. Ellenby, and B. Goldstein, "An automatic beat detection algorithm for pressure signals," *Biomedical Engineering, IEEE Transactions on*, vol. 52, no. 10, pp. 1662–1670, 2005.
- [60] V. X. Afonso, W. J. Tompkins, T. Nguyen, S. Trautmann, and S. Luo, "Filter bank-based processing of the stress ECG," in *Engineering in Medicine and Biology Society, 1995., IEEE 17th Annual Conference*, vol. 2. IEEE, 1995, pp. 887–888.
- [61] V. X. Afonso, W. J. Tompkins, T. Q. Nguyen, K. Michler, and S. Luo, "Comparing stress ECG enhancement algorithms," *Engineering in Medicine and Biology Magazine, IEEE*, vol. 15, no. 3, pp. 37–44, 1996.
- [62] H. Alemzadeh, Z. Jin, Z. Kalbarczyk, and R. K. Iyer, "An embedded re-configurable architecture for patient-specific multi-paramater medical monitoring," in *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*. IEEE, 2011, pp. 1896–1900.
- [63] S. Karpagachelvi, M. Arthanari, and M. Sivakumar, "ECG feature extraction techniques - A survey approach," *International Journal of Computer Science and Information Security*, vol. 8, no. 1, pp. 76–80, Apr. 2010.
- [64] J. Blacher, J. A. Staessen, X. Girerd, J. Gasowski, L. Thijs, L. Liu, J. G. Wang, R. H. Fagard, and M. E. Safar, "Pulse pressure not mean pressure determines cardiovascular risk in older hypertensive patients," *Archives of Internal Medicine*, vol. 160, no. 8, pp. 1085–1089, 2000.

- [65] R. K. Reddy, M. J. Gleva, B. E. Gliner, G. L. Dolack, P. J. Kudenchuk, J. E. Poole, and G. H. Bardy, "Biphasic transthoracic defibrillation causes fewer ECG ST-segment changes after shock," *Annals of Emergency Medicine*, vol. 30, no. 2, pp. 127–134, 1997.
- [66] R. Schoenberg, D. Sands, and C. Safran, "Making ICU alarms meaningful: A comparison of traditional vs. trend-based algorithms," in *Proceedings of the AMIA Symposium*. American Medical Informatics Association, 1999, p. 379.
- [67] C. L. Tsien, "Event discovery in medical time-series data." in *Proceedings of the AMIA Symposium*. American Medical Informatics Association, 2000, p. 858.
- [68] D. Apiletti, E. Baralis, G. Bruno, and T. Cerquitelli, "Real-time analysis of physiological data to support medical applications," *Information Technology in Biomedicine, IEEE Transactions on*, vol. 13, no. 3, pp. 313–321, 2009.
- [69] U. R. Acharya, K. P. Joseph, N. Kannathal, C. M. Lim, and J. S. Suri, "Heart rate variability: A review," *Medical and Biological Engineering and Computing*, vol. 44, no. 12, pp. 1031–1051, 2006.
- [70] T. He, G. Clifford, and L. Tarassenko, "Application of independent component analysis in removing artefacts from the electrocardiogram," *Neural Computing & Applications*, vol. 15, no. 2, pp. 105–116, 2006.
- [71] J. Wang, "A new method for evaluating ECG signal quality for multi-lead arrhythmia analysis," in *Computers in Cardiology, 2002*. IEEE, 2002, pp. 85–88.
- [72] A. V. Deshmane, "False arrhythmia alarm suppression using ECG, ABP, and photoplethysmogram," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.
- [73] N. Kannathal, U. R. Acharya, E. Ng, S. Krishnan, L. C. Min, and S. Laxminarayan, "Cardiac health diagnosis using data fusion of cardiovascular and haemodynamic signals," *Computer Methods and Programs in Biomedicine*, vol. 82, no. 2, pp. 87–96, 2006.
- [74] L. Tarassenko, A. Hann, A. Patterson, E. Braithwaite, K. Davidson, V. Barber, and D. Young, "Biosign: Multi-parameter monitoring for early warning of patient deterioration," in *Proceedings of the 3rd IEEE International Seminar on Medical Applications of Signal Processing*, 2005, pp. 71–76.

- [75] Q. Li and G. D. Clifford, "Signal quality and data fusion for false alarm reduction in the intensive care unit," *Journal of Electrocardiology*, vol. 45, no. 6, pp. 596–603, 2012.
- [76] H. Kim, R. F. Yazicioglu, S. Kim, N. Van Helleputte, A. Artes, M. Konijnenburg, J. Huisken, J. Penders, and C. Van Hoof, "A configurable and low-power mixed signal SoC for portable ECG monitoring applications," in *VLSI Circuits (VLSIC), 2011 Symposium on*. IEEE, 2011, pp. 142–143.
- [77] M. Chang, Z. Lin, C. Chang, H. L. Chan, and W. S. Feng, "Design of a system-on-chip for ECG signal processing," in *Circuits and Systems, 2004. Proceedings. The 2004 IEEE Asia-Pacific Conference on*, vol. 1. IEEE, 2004, pp. 441–444.
- [78] K. Patel, S. McGettrick, and C. J. Bleakley, "Syscore: A coarse grained reconfigurable array architecture for low energy biosignal processing," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 109–112.
- [79] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "ULP-SRP: Ultra low-power Samsung reconfigurable processor for biomedical applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, p. 22, 2014.
- [80] M. Cvikl and A. Zemva, "FPGA-oriented HW/SW implementation of ECG beat detection and classification algorithm," *Digital Signal Processing*, vol. 20, no. 1, pp. 238–248, 2010.
- [81] I. Romero, B. Grundlehner, J. Penders, J. Huisken, and Y. H. Yassin, "Low-power robust beat detection in ambulatory cardiac monitoring," in *Biomedical Circuits and Systems Conference, 2009. BioCAS 2009. IEEE*. IEEE, 2009, pp. 249–252.
- [82] J. Liang and Y. Wu, "Wireless ECG monitoring system based on OMAP," in *Computational Science and Engineering, 2009. CSE'09. International Conference on*, vol. 2. IEEE, 2009, pp. 1002–1006.
- [83] T. Desel, T. Reichel, S. Rudischhauser, and H. Hauer, "A CMOS nine channel ECG measurement IC," in *ASIC, 1996., 2nd International Conference on*. IEEE, 1996, pp. 115–118.
- [84] S. Borromeo, C. Rodriguez-Sanchez, F. Machado, J. A. Hernandez-Tamames, and R. de la Prieta, "A reconfigurable, wearable, wireless ECG system," in *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*. IEEE, 2007, pp. 1659–1662.

- [85] C. Ghule, D. Wakde, G. Viridi, and N. R. Khodke, "Design of portable ARM processor based ECG module for 12 lead ECG data acquisition and analysis," in *Biomedical and Pharmaceutical Engineering, 2009. ICBPE'09. International Conference on.* IEEE, 2009, pp. 1–8.
- [86] Z. Li and Y. Jiang, "An optimal adaptive checkpoint strategy for DMR with energy-aware," in *Parallel and Distributed Computing, Applications and Technologies, 2006. PDCAT'06. Seventh International Conference on.* IEEE, 2006, pp. 535–538.
- [87] V. Petrovic, M. Ilic, G. Schoof, and Z. Stamenkovic, "Design methodology for fault tolerant ASICs," in *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2012 IEEE 15th International Symposium on.* IEEE, 2012, pp. 8–11.
- [88] G. Schoof, M. Methfessel, and R. Kraemer, "Fault-tolerant ASIC design for high system dependability," in *Advanced Microsystems for Automotive Applications 2009.* Springer, 2009, pp. 369–382.
- [89] T. Schweizer, P. Schlicker, S. Eisenhardt, T. Kuhn, and W. Rosenstiel, "Low-cost TMR for fault-tolerance on coarse-grained reconfigurable architectures," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on.* IEEE, 2011, pp. 135–140.
- [90] T. Schweizer, A. Kuster, S. Eisenhardt, T. Kuhn, and W. Rosenstiel, "Using run-time reconfiguration to implement fault-tolerant coarse grained reconfigurable architectures," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International.* IEEE, 2012, pp. 320–327.
- [91] C. Carmichael, "Triple module redundancy design techniques for Virtex FPGAs," *Xilinx Application Note XAPP197*, vol. 1, 2001.
- [92] S. D'Angelo, C. Metra, S. Pastore, A. Pogutz, and G. R. Sechi, "Fault-tolerant voting mechanism and recovery scheme for TMR FPGA-based systems," in *Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings., 1998 IEEE International Symposium on.* IEEE, 1998, pp. 233–240.
- [93] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Efficiently supporting fault-tolerance in FPGAs," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays.* ACM, 1998, pp. 105–115.
- [94] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic fault tolerance in FPGAs via partial reconfiguration," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on.* IEEE, 2000, pp. 165–174.

- [95] S. Eisenhardt, A. Kuster, T. Schweizer, T. Kuhn, and W. Rosenstiel, "Spatial and temporal data path remapping for fault-tolerant coarse-grained reconfigurable architectures," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2011 IEEE International Symposium on. IEEE, 2011, pp. 382–388.
- [96] L. Anghel, D. Alexandrescu, and M. Nicolaidis, "Evaluation of a soft error tolerance technique based on time and/or space redundancy," in *Integrated Circuits and Systems Design, 2000. Proceedings. 13th Symposium on*. IEEE, 2000, pp. 237–242.
- [97] K. Mohanram and N. A. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *2013 IEEE International Test Conference (ITC)*. IEEE, 2003, pp. 893–893.
- [98] S. M. Jafri, S. J. Piestrak, O. Sentieys, and S. Pillement, "Design of a fault-tolerant coarse-grained reconfigurable architecture: A case study," in *Quality Electronic Design (ISQED)*, 2010 11th International Symposium on. IEEE, 2010, pp. 845–852.
- [99] M. M. Azeem, S. J. Piestrak, O. Sentieys, and S. Pillement, "Error recovery technique for coarse-grained reconfigurable architectures," in *Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2011 IEEE 14th International Symposium on. IEEE, 2011, pp. 441–446.
- [100] M. Pflanz and H. T. Vierhaus, "Online check and recovery techniques for dependable embedded processors," *IEEE Micro*, vol. 21, no. 5, pp. 24–40, 2001.
- [101] "OpenCore: 16-bit MIPS processor with five-stage pipeline," 2013. [Online]. Available: [http://opencores.org/project,mips\\_16/](http://opencores.org/project,mips_16/)
- [102] M. Y. Hsiao, "A class of optimal minimum odd-weight-column SECDED codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.
- [103] "Google Android software development kit," Sep. 2014. [Online]. Available: <https://developer.android.com/sdk/index.html?hl=i/>
- [104] "Qualcomm Trepn profiler for Android," Sep. 2014. [Online]. Available: <https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler/>
- [105] "NanGate FreePDK45 open cell library (45 nm)," May 2014. [Online]. Available: [http://www.nangate.com/?page\\_id=2325/](http://www.nangate.com/?page_id=2325/)

- [106] “Synopsys Generic Memory Compiler (University program),” Oct. 2014. [Online]. Available: <http://www.synopsys.com/COMMUNITY/UNIVERSITYPROGRAM/Pages/generic-memory-compiler.aspx/>
- [107] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, “Crashtest: A fast high-fidelity FPGA-based resiliency analysis framework,” in *Computer Design, 2008. ICCD 2008. IEEE International Conference on*. IEEE, 2008, pp. 363–370.